Program Abstraction in a Higher-Order Logic Framework

Marco Benini Sara Kalvala Dirk Nowotka

Department of Computer Science
University of Warwick, Coventry, CV4 7AL, United Kingdom

Abstract. We present a hybrid approach to program verification: a higher-order logic, used as a specification language, and a human-driven proof environment, with a process-algebraic engine to allow the use of process simulation as an abstraction technique. The domain of application is the validation of object code, and our intent is to adapt and mix existing formalisms to make the verification of representative programs possible. In this paper, we describe the logic in question and an underlying semantics given in terms of a process algebra.

1 Introduction

Software validation through formal verification has been a topic of much research over the years, but widespread adoption of developed techniques is still elusive. One may attribute this to many factors—two of them being the difficulty in performing the verification and the relevance of the formal proof to actual trustworthiness of the software product.

Most of the techniques being developed have approached the problem by incorporating the semantics of the high-level language of the original program, proving a correspondence with the specification. However, this approach assumes the correct implementation of an abstract program (the high-level programming language) on some hardware. One can either rely on that implementation or show that the translation of high-level code to machine code by the involved compiler is sound. But, doing that is not trivial. Real compilers do not work by incorporating so called correctness-preserving transformations, but by invoking many optimising heuristics and practical but informal knowledge. One may therefore question the relevance of the verification exercise to the correctness of the actual code that is to be run. This does not invalidate the verification of high-level programs, as many design errors can still be found and corrected.

We believe that there is scope for verifying the compiled code itself, which relies only on the correctness of used hardware directly and no program code transformations. This gives a better assurance about the overall system, which is after all the object one would like to have confidence in. That approach is called *object code verification*.

A specification language has to be fixed. We decided to use logical specifications, and as such *higher-order logic*. This decision distinguishes our approach

from others like Pavey and Winsborrow [9], who used a rather informal mapping of program code into MALPAS Intermediate Language, in mathematical rigour and Yu [11], who used the quantifier free, first-order logic of Nqthm, in expressiveness. Our decision was also influenced by the availability of automated theorem provers for higher-order logic like HOL [2] and Isabelle/HOL [6, 8].

However, there are difficulties in the verification of programs written in a low-level language. Such a language is, in a sense, further away from the specification language because it has to deal with a more concrete machine, considering registers, limited memory, and so on. There is a vast shift in granularity between code and specification. This gap strongly suggests the application of abstraction techniques on the data as well as control structure of a program.

The concept of simulation [3] and observational equivalence [5] in the framework of process algebra gives us a well-developed tool for abstraction. We use a variation of a traditional process algebra (Milner's CCS [5]), which is called ωCCS. In this setting—HOL and ωCCS—the verification effort would consist of abstraction steps interspersed with interesting *inference* steps where correctness properties are derived. So, one approach that can be used for structuring a verification task is to separate the inference steps from the abstraction steps, and support each of them in an optimal manner.

In this paper, we describe a system which allows the use of both inference in higher-order logic and abstraction in process algebra while still maintaining consistency. The use of this formalism allows us to structure the verification task as illustrated in Fig. 1. The left-hand side of the picture (grey arrows) is the way we assume the software to be developed, i.e., starting from a specification a program (in some high-level language) has been written, and the object code was generated from that. Formalising the specification, we obtain an higher-order formula. From the object code, by means of our tool, another higher-order formula is constructed; we call this formula the *representation* of our program. The usual process of formal verification (solid arrows in Fig. 1) is to try to *infer* the validity of the formalised specification assuming the representation.

Our tool provides the instruments to perform such deductions. But it also provides an abstraction mechanism based on ωCCS (right-hand side of Fig. 1): translating a program representation into an ωCCS expression, we can refine it, and we get as a result an abstracted ωCCS expression, we can then translate back to higher-order logic, and this is another representation of the original program, not equivalent to the original one, but preserving enough information so to ensure that, if we are able to deduce the specification from it, then there is (a more complex) derivation of the same specification from the original representation.

We will illustrate our approach by first introducing ωCCS, our process algebra, and showing how we model abstraction in this system. We then describe the logic used, which provides the syntax and a proof interface.

Our emphasis is on showing the correctness of the process algebraic formalism with respect to the logical representation, so we illustrate the correspondence between these two formalisms.

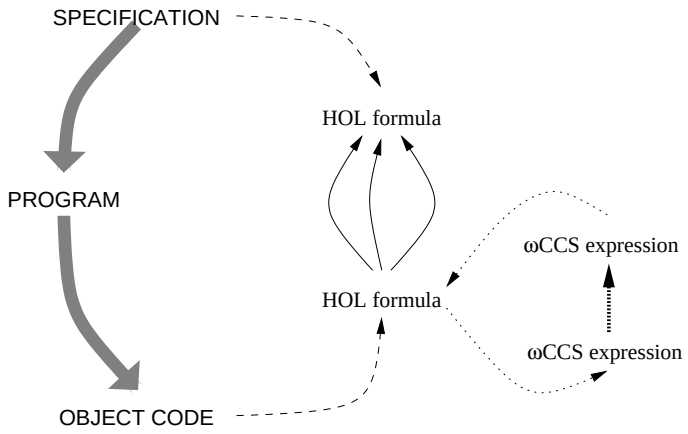

Fig. 1. Verification integrating inference and abstraction.

Finally, we sketch a practical embodiment of this system as an extension to the Isabelle/HOL theorem prover.

A particular assembly language has been fixed to allow the comparison of our work with others in the field (e.g. [11]). We use the Motorola M68000 architecture here.

2 An algebra for processes

In this section we give a detailed overview of ωCCS, a variant of the Calculus of Communicating Systems (CCS). The motivation for this variant is given by explaining the main difficulty we have in using the standard CCS for our verification endeavours, namely in the characterisation of divergence.

2.1 Basic notions

The CCS process algebra is an intuitive framework for describing processes. Our extension is based on *message passing CCS* [5, pp. 53–56], which models actions as objects capable of reading a value and co-actions as objects which are able to send a value.

As in CCS, we assume as given two distinct sets, Actions and Values, which are used to build processes, following the syntax in Fig. 2.

We refer to literature [3–5] for a detailed explanation of the intuitive and formal meaning of process constructors. The only syntactical differences we introduced with respect to CCS are:

- renaming involves values as well as actions;
- as in ACP [1], we have a sequential composition operator: $P\,;;\,Q$ should be read as "P then Q", and it is the process which behaves like P until P terminates, if ever, then it behaves like Q.

$$
\begin{array}{rcl}
\langle Process\rangle & ::= & \underline{\langle Action\rangle} \,.\, \langle Value \longrightarrow Process\rangle \\
& | & \overline{\langle Action\rangle}(\langle Value\rangle) \,.\, \langle Process\rangle \\
& | & \tau \,.\, \langle Process\rangle \\
& | & \sum \langle Process\ set\rangle \\
& | & \langle Process\rangle \mid \langle Process\rangle \\
& | & \mu\, \langle Process \longrightarrow Process\rangle \\
& | & \langle Process\rangle \setminus \langle Action\ set\rangle \\
& | & \langle Process\rangle[\langle (Action \longrightarrow Action) \times (Value \longrightarrow Value)\rangle] \\
& | & \langle Process\rangle \,;; \langle Process\rangle
\end{array}
$$

Fig. 2. Syntax of ωCCS processes.

In the usual way, a transition relation is defined for ωCCS. The basic rules are shown in Fig. 3.

Since we are interested in observing only visible actions, because they corresponds in our frame to the execution of instructions, we use the so called *observational transition relation*:

$$P \stackrel{\alpha}{\Longrightarrow} Q \text{ iff } \alpha \neq \tau \wedge \exists P', Q'.P \stackrel{\tau^*}{\longrightarrow} P' \wedge P' \stackrel{\alpha}{\longrightarrow} Q' \wedge Q' \stackrel{\tau^*}{\longrightarrow} Q \ .$$

The properties about these relations, not involving simulations, are the same as for CCS.

In our implementation, we designed an Isabelle/HOL theory that, representing processes as a datatype and defining by co-induction [7] the transition relations, give the possibility to prove all the lemmas and theorems concerning these basic definitions. We proved most of them.

2.2 Simulation and divergence

As we said in the introduction, our goal is to use ωCCS to abstract over program representations. Let us suppose that a program is represented by the process P, we say that another process Q is an *abstraction* over P, if it is able to behave like P.

The process algebraic counterpart of this notion is *simulation*. Since we focus our attention to visible actions, a good candidate for modeling our notion of abstraction seems to be *weak simulation* [5]:

$$P \preccurlyeq Q \quad \text{iff} \quad \forall \beta, P'.P \stackrel{\beta}{\longrightarrow} P' \rightarrow \exists Q'.Q \stackrel{\beta}{\Longrightarrow} Q' \wedge P' \preccurlyeq Q' \ .$$

It is easy to show that $\preccurlyeq$ is a precongruence [5, 10] with respect to all operators of ωCCS, with the exception of sequence. We will return to this point later.

Unfortunately, our intuitive notion of behaviour for a program does not correspond to the idea of behaviour embodied into the weak simulation. A behaviour, for weak simulation, is a sequence of visible actions a process may perform, and

$$\frac{}{\alpha\,.\,(\lambda w.P(w)) \xrightarrow{\alpha(v)} P(v)} \qquad \frac{}{\overline{\alpha}(v)\,.\,P \xrightarrow{\overline{\alpha}(v)} P} \qquad \frac{}{\tau\,.\,P \xrightarrow{\tau} P}$$

$$\frac{P \xrightarrow{\beta} Q}{\sum(\{P\} \cup \Xi) \xrightarrow{\beta} Q}$$

$$\frac{P \xrightarrow{\beta} Q}{P \mid R \xrightarrow{\beta} Q \mid R} \qquad \frac{P \xrightarrow{\beta} Q}{R \mid P \xrightarrow{\beta} R \mid Q}$$

$$\frac{P \xrightarrow{\alpha(v)} P' \quad Q \xrightarrow{\overline{\alpha}(v)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\frac{P(\mu\,(\lambda w.P(w))) \xrightarrow{\beta} Q}{\mu\,(\lambda w.P(w)) \xrightarrow{\beta} Q}$$

$$\frac{P \xrightarrow{\alpha(v)} Q \quad \alpha \notin \Delta}{P \setminus \Delta \xrightarrow{\alpha(v)} Q \setminus \Delta} \qquad \frac{P \xrightarrow{\overline{\alpha}(v)} Q \quad \alpha \notin \Delta}{P \setminus \Delta \xrightarrow{\overline{\alpha}(v)} Q \setminus \Delta} \qquad \frac{P \xrightarrow{\tau} Q}{P \setminus \Delta \xrightarrow{\tau} Q \setminus \Delta}$$

$$\frac{P \xrightarrow{\alpha(v)} Q}{P[\Phi] \xrightarrow{(\Phi(\alpha))(\Phi(v))} Q[\Phi]} \qquad \frac{P \xrightarrow{\overline{\alpha}(v)} Q}{P[\Phi] \xrightarrow{\overline{(\Phi(\alpha))}(\Phi(v))} Q[\Phi]} \qquad \frac{P \xrightarrow{\tau} Q}{P[\Phi] \xrightarrow{\tau} Q[\Phi]}$$

$$\frac{P \xrightarrow{\beta} Q}{P\,;;\,R \xrightarrow{\beta} Q\,;;\,R}$$

Where: v, w are variables ranging over values,
X is a variable ranging over processes,
P, Q, R are processes,
α is any action except τ,
β is of the form $\alpha(v)$, $\overline{\alpha}(v)$ or τ,
Δ is any set of actions,
Ξ is any set of processes.

Moreover, we know that $\sum\emptyset$ is the termination process, that $\sum\{P\} \equiv P$, that $\overline{\overline{\alpha}}(v) \equiv \alpha(v)$, and that $\sum\emptyset\,;;\,P \equiv P$. Since in renaming, Φ denotes a pair of functions $\langle\phi, \theta\rangle$,

$$(\Phi(\alpha))(\Phi(v)) \equiv (\phi(\alpha))(\theta(v)) \ .$$

Fig. 3. The basic transition relation for ωCCS.

a way to read $P \preccurlyeq Q$ is that "for every sequence of visible actions P can perform, the same sequence is a possible behaviour for Q".

Suppose now that P represent a program which is in an unrecoverable deadlock situation: it continues to request a service which is permanently unavailable. From the observer point of view, there is no visible action, but he is able to recognise that the program is "doing nothing", but it has not terminated its execution. Suppose $Q \equiv \sum \emptyset$, that means Q is the program which reached happily its last instruction, so it has finished its job with no troubles. Again, from the observer point of view, Q is unable to perform any visible action, but, this time, the observer knows that Q has terminated its execution.

But according to the definition of weak simulation, $P \preccurlyeq Q$, so we have lost an essential piece of information regarding the behaviour of P: in general, it is easy to figure out examples which show that situations like *starvation* and *deadlocks* are not preserved by weak simulation. Obviously, there is no hope to use weak simulation as an abstraction tool in a formal verification task, since, most of the time, the verifier must check exactly for the kind of faults which are not preserved by the $\preccurlyeq$ relation.

From a formal point of view, we want to preserve *divergence*:

$$\text{diverge}(P) \quad \text{iff} \quad \exists f. f(0) = P \wedge \forall n. f(n) \xrightarrow{\tau} f(n+1) \ .$$

A process is said to be divergent if it is able to produce an infinite trace which is composed only by τ actions.

Our main abstraction tool, we call it *div-simulation* is defined as:

$$P \overset{div}{\preccurlyeq} Q \quad \text{iff} \quad P \preccurlyeq Q \wedge (\text{diverge}(P) \rightarrow \text{diverge}(Q)) \ .$$

The main properties regarding this notion are shown in Fig. 4.

Compared to other results in literature (e.g., [1, 10]), our notion is more primitive and not so attractive from the mathematical point of view, since div-simulation is not a precongruence: the rule

$$\frac{P \overset{div}{\preccurlyeq} Q}{P \,;; R \overset{div}{\preccurlyeq} Q \,;; R} \tag{1}$$

does not hold; if $P \equiv \sum \emptyset$, $Q \equiv \overline{\alpha}(v) \,.\, \sum \emptyset$, $R \equiv \overline{\gamma}(v) \,.\, \sum \emptyset$, where α and γ are different actions, it follows that $P \preccurlyeq Q$ and $\neg\text{diverge}(P)$, so $P \overset{div}{\preccurlyeq} Q$, but $P \,;; R \xrightarrow{\gamma(v)} \sum \emptyset$ while $Q \,;; R \overset{\gamma(v)}{\not\longrightarrow} X$, for any process X, so $P \,;; R \not\preccurlyeq Q \,;; R$.

In this respect, it becomes evident that even weak simulation is not a precongruence when a sequencing operator is added to the process algebra. This result is not new, since it is already discussed in [5, Section 9.2], but the solution proposed there, i.e., using a special action which denotes termination, is impractical in most verification tasks, since it forces the verifier to use different representations for subsections of code. As an example, if we want to prove correctness of a loop, the algebraic representation must produce a "termination" action as

$$\frac{P \overset{div}{\preccurlyeq} Q}{\tau \,.\, P \overset{div}{\preccurlyeq} \tau \,.\, Q} \qquad \frac{P \overset{div}{\preccurlyeq} Q}{\overline{\alpha}(v) \,.\, P \overset{div}{\preccurlyeq} \overline{\alpha}(v) \,.\, Q} \qquad \frac{\forall w.P(w) \overset{div}{\preccurlyeq} Q(w)}{\alpha \,.\, P \overset{div}{\preccurlyeq} \alpha \,.\, Q}$$

$$\frac{P \overset{div}{\preccurlyeq} Q}{\sum \Xi \cup \{P\} \overset{div}{\preccurlyeq} \sum \Xi \cup \{Q\}} \qquad \frac{P \overset{div}{\preccurlyeq} P' \quad Q \overset{div}{\preccurlyeq} Q'}{P \mid Q \overset{div}{\preccurlyeq} P' \mid Q'}$$

$$\frac{P \overset{div}{\preccurlyeq} Q}{P \setminus \Delta \overset{div}{\preccurlyeq} Q \setminus \Delta} \qquad \frac{P \overset{div}{\preccurlyeq} Q}{P[\Phi] \overset{div}{\preccurlyeq} Q[\Phi]}$$

$$\frac{P(\mu\, P) \overset{div}{\preccurlyeq} Q(\mu\, Q)}{\mu\, P \overset{div}{\preccurlyeq} \mu\, Q} \qquad \frac{P \overset{div}{\preccurlyeq} Q}{R \,;; P \overset{div}{\preccurlyeq} R \,;; Q}$$

Fig. 4. Basic properties of div-simulation.

the last step of the loop computation; if this loop is part of a bigger code, we cannot simply immerse it by using the sequence operator, so we are forced to use something like the *Before* operator [5, p. 173]. This is not satisfactory since we must know in advance where we plan to divide the code for verificational purposes, and this is not true in real practice.

Anyway, our notion is powerful enough to give us the right instrument to perform abstraction. In fact a restricted version of (1) holds:

$$\frac{P \overset{div}{\preccurlyeq} Q \quad Q \overset{div}{\preccurlyeq} P}{P \,;; R \overset{div}{\preccurlyeq} Q \,;; R}$$

Most of the time, this rule is just what we need for our purposes. We also note that if P is *finitary*, i.e., if P always reduces to $\sum \emptyset$, then $P \,;; Q$ can be converted to an equivalent process R, where sequencing is substituted with prefixing and summations.

Our implementation of ωCCS in Isabelle/HOL provides the notion of divergence, of weak simulation and of div-simulation, along with all lemmas presented in this section, plus many others. The coding of these notions is completely standard, using the coinduction package [7], and all proofs are carried on following the guidelines traced in [5].

2.3 An illustrating example

The main motivation for introducing a process algebra into our verification methodology is to cope with the interrupt system of a machine, or more generally, to cope with processes running in parallel. This is necessary when we want to verify programs for a "real-world" environment.

Let us have a look at interrupts triggered by the program itself, like for instance operating system calls (see Fig. 5).

Program	**Operating System**

A
syscall 7
B
	`case` *interrupt* `of`
	1: I_1
	...
	7: I_7
	...
	`esac`
	`return`

Fig. 5. Software interrupts

This kind of interaction of a program with its environment does not necessarily need a concurrent model. It could simply be the sequential composition of the program parts, A and B, with the interrupt routine, I_7, which would be the way one would treat such a system following [11].

$$Program \stackrel{\text{def}}{=} A \,;; I_7 \,;; B$$

Nevertheless, we prefer the following variant:

$$Prog \stackrel{\text{def}}{=} ((A \,;; \overline{syscall}(7) \,.\, rti \,.\, (\lambda State.B)) \mid OS) \setminus \{syscall,\, rti\}$$
$$OS \stackrel{\text{def}}{=} syscall \,.\, \lambda i.I_i \,;; \overline{rti}(State) \,.\, OS$$

Though it looks more complicated, this new approach allows us to handle external interrupts, that is, events that can interrupt the program at any possible time and change the state of the machine. Such a system could hardly be modeled with the first approach above.

The whole system, i.e. the program and its run-time environment, would then look like this:

$$System \stackrel{\text{def}}{=} (Prog \mid (Proc(InitState) \mid Env)) \setminus \{rd,\, wr,\, int,\, ret,\, rd_{int},\, wr_{int}\}$$

where $InitState$ is the initial state. The process components are defined below.

$$\begin{aligned}
Prog &\stackrel{\text{def}}{=} rd\,.\,(\lambda State.(\overline{wr}(State')\,.\,Prog)) \\
Proc(State) &\stackrel{\text{def}}{=} \overline{rd}(State)\,.\,Proc(State) + wr\,.\,(\lambda State'.Proc(State')) \\
&\quad + int\,.\,Proc'(State) \\
Proc'(State) &\stackrel{\text{def}}{=} \overline{rd_{int}}(State)\,.\,Proc'(State) + wr_{int}\,.\,(\lambda State'.Proc'(State')) \\
&\quad + \overline{ret}\,.\,Proc(State) \\
Env &\stackrel{\text{def}}{=} \overline{int}\,.\,Env + rd_{int}\,.\,(\lambda State.(\overline{wr_{int}}(State')\,.\,Env)) \\
&\quad + ret\,.\,Env
\end{aligned}$$

Every component of the system can now be subject to abstraction as long as its interaction with the system is not modified. The environment, for instance, could be altered in a way that the machine state is not changed by an interrupt routine.

$$System' \stackrel{\text{def}}{=} (Prog \mid (Proc''(InitState) \mid Env')) \setminus \{rd,\, wr,\, int,\, ret\}$$

Where we have:

$$\begin{aligned}
Proc''(State) &\stackrel{\text{def}}{=} \overline{rd}(State)\,.\,Proc''(State) + wr\,.\,(\lambda State'.Proc''(State')) \\
&\quad + int\,.\,\overline{ret}\,.\,Proc''(State) \\
Env' &\stackrel{\text{def}}{=} \overline{int}\,.\,Env' + ret\,.\,Env'
\end{aligned}$$

Since

$$(Proc(InitState) \mid Env) \overset{div}{\preccurlyeq} (Proc''(InitState) \mid Env')$$

and obviously $Prog \overset{div}{\preccurlyeq} Prog$, we have $System \overset{div}{\preccurlyeq} System'$. It is easy to see that our (simple) model can engage into starvation of the program, which is the case when interrupts occur continuously. That is why the following would be a wrong abstraction.

$$\begin{aligned}
System'' &\stackrel{\text{def}}{=} (Prog \mid (Proc'''(InitState) \mid Env'')) \setminus \{rd,\, wr\} \\
Proc'''(State) &\stackrel{\text{def}}{=} \overline{rd}(State)\,.\,Proc'''(State) + wr\,.\,(\lambda State'.Proc'''(State')) \\
Env'' &\stackrel{\text{def}}{=} \sum \emptyset
\end{aligned}$$

Because that abstraction does *not* preserve divergence.

$$(Proc(InitState) \mid Env) \overset{div}{\not\preccurlyeq} (Proc'''(InitState) \mid Env'')$$

A termination result for $Prog$ in $System''$ does not ensure us that $Prog$ actually terminates in $System$, too.

3 Program representation in higher-order logic and ωCCS

The standard syntax used for higher-order logic is used here. However, when working with Isabelle/HOL the syntax is, of course, more "ASCII like", but the translation is straightforward and should pose no problems in understanding (see also [8]).

Terms are defined in a way that allows arithmetic on integer numbers. We also define functions to model registers (D_0, ... , D_7, A_0, ... A_7), the program counter (PC), and memory (Mem). These functions take a term, representing time, to a term, representing a value. (Mem also takes a term, representing an address, as argument.) Flags, i.e., elements of the condition code register, are represented by predicates (C, N, V, X, Z) taking a term, that represents time, as argument. See Fig. 6 and Fig. 7.

$$
\begin{aligned}
\langle Term\rangle ::= {} & 1 \mid 2 \mid 3 \mid \ldots \\
\mid {} & \langle Variable\rangle \\
\mid {} & \langle Term\rangle + \langle Term\rangle \\
\mid {} & \langle Term\rangle - \langle Term\rangle \\
\mid {} & \langle Term\rangle \times \langle Term\rangle \\
\mid {} & \langle Term\rangle \div \langle Term\rangle \\
\mid {} & \langle Term\rangle^{\langle Term\rangle} \\
\mid {} & \langle Term\rangle \bmod \langle Term\rangle \\
\mid {} & \mathrm{PC}(\langle Term\rangle) \\
\mid {} & \mathrm{D}_0(\langle Term\rangle) \mid \mathrm{D}_1(\langle Term\rangle) \mid \ldots \mid \mathrm{D}_7(\langle Term\rangle) \\
\mid {} & \mathrm{A}_0(\langle Term\rangle) \mid \mathrm{A}_1(\langle Term\rangle) \mid \ldots \mid \mathrm{A}_7(\langle Term\rangle) \\
\mid {} & \mathrm{Mem}_{\langle Term\rangle}(\langle Term\rangle)
\end{aligned}
$$

Fig. 6. Term syntax.

$$
\begin{aligned}
\langle Formula\rangle ::= {} & \mathrm{true} \mid \mathrm{false} \\
\mid {} & \mathrm{C}(\langle Term\rangle) \mid \mathrm{N}(\langle Term\rangle) \mid \mathrm{V}(\langle Term\rangle) \mid \mathrm{X}(\langle Term\rangle) \mid \mathrm{Z}(\langle Term\rangle) \\
\mid {} & \langle Term\rangle = \langle Term\rangle \mid \langle Term\rangle < \langle Term\rangle \mid \langle Term\rangle \leq \langle Term\rangle \\
\mid {} & \neg\langle Formula\rangle \mid \langle Formula\rangle \wedge \langle Formula\rangle \mid \langle Formula\rangle \vee \langle Formula\rangle \\
\mid {} & \langle Formula\rangle \rightarrow \langle Formula\rangle \mid \langle Formula\rangle \leftrightarrow \langle Formula\rangle \\
\mid {} & \exists\langle Variable\rangle.\langle Formula\rangle \mid \forall\langle Variable\rangle.\langle Formula\rangle
\end{aligned}
$$

Fig. 7. Formula syntax.

The meaning of the introduced functions is as one would expect and is best illustrated with an example program.

The instructions of a small program in assembly language, shown in Fig. 8, illustrate the program representation in higher-order logic. Every line in the table is numbered and shows one assembly instruction followed by its representation as a logical formula. This representation has already been manipulated: if one knows the specification, it is straightforward to cut irrelevant references to memory/registers from the representation. Since this procedure has been automated in our system and it does not affect the verification validity (but improves performance and readability), we assume this simplification being done implicitly.

1:	`MOVE #0, D1`	$\mathrm{PC}(t) = 1 \wedge \mathrm{PC}(t+1) = 2 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t) \wedge \mathrm{D}_1(t+1) = 0$
2:	`MOVE D1, D2`	$\mathrm{PC}(t) = 2 \wedge \mathrm{PC}(t+1) = 3 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t)$ $\wedge\, \mathrm{D}_1(t+1) = \mathrm{D}_1(t) \wedge \mathrm{D}_2(t+1) = \mathrm{D}_1(t)$
3:	`MULT D2, D2`	$\mathrm{PC}(t) = 3 \wedge \mathrm{PC}(t+1) = 4 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t)$ $\wedge\, \mathrm{D}_1(t+1) = \mathrm{D}_1(t) \wedge \mathrm{D}_2(t+1) = \mathrm{D}_2(t)^2$
4:	`CMP D2, D0`	$\mathrm{PC}(t) = 4 \wedge \mathrm{PC}(t+1) = 5 \wedge (\mathrm{N}(t+1) \leftrightarrow \mathrm{D}_2(t) \leq \mathrm{D}_0(t))$ $\wedge\, \mathrm{D}_0(t+1) = \mathrm{D}_0(t) \wedge \mathrm{D}_1(t+1) = \mathrm{D}_1(t)$
5:	`BGT 8`	$\mathrm{PC}(t) = 5 \wedge (\neg\mathrm{N}(t) \rightarrow \mathrm{PC}(t+1) = 6) \wedge (\mathrm{N}(t) \rightarrow \mathrm{PC}(t+1) = 8)$ $\wedge\, \mathrm{D}_0(t+1) = \mathrm{D}_0(t) \wedge \mathrm{D}_1(t+1) = \mathrm{D}_1(t)$
6:	`ADD #1, D1`	$\mathrm{PC}(t) = 6 \wedge \mathrm{PC}(t+1) = 7 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t)$ $\wedge\, \mathrm{D}_1(t+1) = 1 + \mathrm{D}_1(t)$
7:	`BRA 2`	$\mathrm{PC}(t) = 7 \wedge \mathrm{PC}(t+1) = 2 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t)$ $\wedge\, \mathrm{D}_1(t+1) = \mathrm{D}_1(t)$
8:	`SUB #1, D1`	$\mathrm{PC}(t) = 8 \wedge \mathrm{PC}(t+1) = 9 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t)$ $\wedge\, \mathrm{D}_1(t+1) = \mathrm{D}_1(t) - 1$

Fig. 8. A small assembly program and its logical representation.

Every instruction depends on a parameter t which represents the time flow. The actual program representation in higher-order logic is

$$\forall t. i_1(t) \vee i_2(t) \vee i_3(t) \vee i_4(t) \vee i_5(t) \vee i_6(t) \vee i_7(t) \vee i_8(t)$$

where $i_j(t)$ stands for the representing formula of instruction j instantiated with t.

This program should calculate the integer square root of a natural number. It terminates with 9 as last program counter value, the argument of the computation is given in register D_0 and the result stands in register D_1. So the program ought to satisfy the following specification:

$$\exists t. \mathrm{PC}(t) = 9 \wedge (\mathrm{D}_1(t))^2 \leq \mathrm{D}_0(t) \wedge \mathrm{D}_0(t) \leq (\mathrm{D}_1(t) + 1)^2. \tag{2}$$

The same program is represented in ωCCS. Due to space limitations that representation is sketched. Nonetheless, some definitions are necessary. Let

$$S = \{\mathrm{PC}, \mathrm{D}_0, \ldots, \mathrm{D}_7, \mathrm{A}_0, \ldots, \mathrm{A}_7, \mathrm{Mem}_i, \mathrm{C}, \mathrm{N}, \mathrm{V}, \mathrm{X}, \mathrm{Z}\} \qquad 0 \leq i < 2^{32}$$

be a set of constants that represent the obvious machine parts. Let a state be of type $S \to \mathbb{N}$. Let a tuple of type $S \times \mathbb{N}$ denote the update of some memory cell/register. Let *overwrite* be a function that takes a state and a set of tuples to a new state. For instance, let $State(\mathrm{PC}) = 1$ and $State(\mathrm{D}_0) = 8$, then $State' = overwrite(State, \{(\mathrm{PC}, 2)\})$ with now $State'(\mathrm{PC}) = 2$ and $State'(\mathrm{D}_0) = 8$.

See Fig. 9 for a representation of the first three assembly lines.

$$
\begin{array}{ll}
1\text{: } \texttt{MOVE \#0, D1} & Instr_1 \stackrel{\text{def}}{=} rd\,.\,\lambda State.\overline{wr}(overwrite \\
& (State, \{(\mathrm{D}_1, 0), (\mathrm{C}, 0), (\mathrm{N}, 0), (\mathrm{V}, 0), (\mathrm{Z}, 1)\}))\,.\,Instr_2 \\
2\text{: } \texttt{MOVE D1, D2} & Instr_2 \stackrel{\text{def}}{=} rd\,.\,\lambda State.\overline{wr}(overwrite \\
& (State, \{(\mathrm{D}_2, State(\mathrm{D}_1)), (\mathrm{C}, 0), \ldots\}))\,.\,Instr_3 \\
3\text{: } \texttt{MULT D2, D2} & Instr_3 \stackrel{\text{def}}{=} rd\,.\,\lambda State.\overline{wr}(overwrite \\
& (State, \{(\mathrm{D}_2, State(\mathrm{D}_2)^2), (\mathrm{C}, 0), \ldots\}))\,.\,Instr_4 \\
\vdots &
\end{array}
$$

Fig. 9. A small assembly program and its ωCCS representation.

The next section illustrates why we took the effort to represent the same program in two theories.

4 Abstraction

As mentioned in Section 1, verification of object code programs is characterised by starting with a program full of many details and eliminating the details which do not affect the properties of interest. By abstraction we mean forgetting about details and focusing on the essentials.

More precisely, if P is a process that represents a program, we can say that the process Q is an abstraction of P if:

- except for "irrelevant" details, Q is able to behave like P.
- the intrinsic properties of P are preserved.

The "irrelevant" details mentioned above have to do with the precise operation of the machine, while an example of an intrinsic property we may want to preserve is the divergence character.

4.1 Abstraction in ωCCS

Recalling from Section 2.2, if $P \overset{div}{\preccurlyeq} Q$ holds, we know that every behaviour satisfying P must satisfy Q and Q cannot be non-divergent if P is divergent.

Let us suppose that P is a process which encodes a program, and let us try to understand what Q is.

Every behaviour of the process P represents, in a formal sense, a computation of the program represented by P. Since Q simulates P, Q is able to perform every computation P may perform. But div-simulation preserves divergence so Q can be divergent even if P is not, and Q must be divergent whenever P is. With this interpretation, the strict nature of programs we chose to model is preserved and Q is a fair model for the program encoded in P, but Q is, in principle, more general since it can exhibit more behaviours than P. So Q, by our intuitive definition of abstraction, is an abstract version of P.

This kind of view can be considered a structural abstraction, something that, enlarging the set of possible behaviours, simplifies the structure of the program.

It is possible to gain data abstraction: if P is a process representing a program, and $P \overset{div}{\preccurlyeq} P[\Phi]$ for a proper Φ, we can use the renaming function to hide, in a fair way, useless variables and parts of the state. Again, having a simulation between P and $P[\Phi]$, we have an abstraction by our intuitive definition.

There are other ways to generate abstractions: essentially, given P, if we have an operation Γ depending on a set Δ of parameters, then $\Gamma_\Delta(P)$ is an abstraction over P if $P \overset{div}{\preccurlyeq} \Gamma_\Delta(P)$. Of course, the interesting part is to discover appropriate Γs in such a way as to generate useful abstractions for P. We have no definite answers in this direction, but it is promising to know that not only is Q an abstraction for P if $P \overset{div}{\preccurlyeq} Q$, but there are no abstractions for P which do not div-simulate P itself. In other words, our framework is rich enough to model every abstraction (in the sense introduced above), but it is not trivial to discriminate useful from useless ones.

From the practical point of view, it is difficult to use div-simulation to perform abstractions. The simplest example we have proved correct with our tool is illustrated in Section 2.3.

As explained in that example, the main reason why we are interested in div-simulation, is to abstract over the whole system, especially for dealing with situations where external interrupts, non predictable faults or asynchronous I/O are involved.

The way to perform abstraction is clear from the example: from the concrete representation P we produce, by means of some reasoning, usually in an heuristic way, an abstract representation Q. Our implementation of ωCCS aids the proof that Q is an abstraction with respect to P, that means, $P \overset{div}{\preccurlyeq} Q$. When the proof is "simple" the ωCCS theory is able to prove that goal by using Isabelle's simplifier, while, in more complex cases, the user has to drive to prover in order to establish the truth of the statement.

4.2 Abstraction in higher-order logic

Our abstraction paradigm could be modeled completely using div-simulation, but this is impractical for two reasons:

– some kinds of abstractions are simple, and used very often;

– the most natural way to conceive some abstractions is quite different from a process algebraic simulation.

For these reasons we enrich our framework in higher-order logic with tools in order to perform common abstractions directly in the logical level (opposing to use the ωCCS representation).

The first tool we provide takes the higher-order logic representation of an object code program along with the specification we want to prove and simplifies the representation itself by removing references to memory and registers which do not appear in the specification. The algorithm to do this simplification uses information about the flow of control of the program to choose what to remove and what to preserve.

In Fig. 8 is shown the result of this simplification with respect to the specification (2). Just as an example, the complete representation of instruction 7 will look like:

$$\begin{aligned}
&\mathrm{PC}(t) = 7 \wedge \mathrm{PC}(t+1) = 2 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t) \wedge \mathrm{D}_1(t+1) = \mathrm{D}_1(t) \\
&\wedge \mathrm{D}_2(t+1) = \mathrm{D}_2(t) \wedge \mathrm{D}_3(t+1) = \mathrm{D}_3(t) \wedge \mathrm{D}_4(t+1) = \mathrm{D}_4(t) \\
&\wedge \mathrm{D}_5(t+1) = \mathrm{D}_5(t) \wedge \mathrm{D}_6(t+1) = \mathrm{D}_6(t) \wedge \mathrm{D}_7(t+1) = \mathrm{D}_7(t) \\
&\wedge \mathrm{A}_0(t+1) = \mathrm{A}_0(t) \wedge \mathrm{A}_1(t+1) = \mathrm{A}_1(t) \wedge \mathrm{A}_2(t+1) = \mathrm{A}_2(t) \\
&\wedge \mathrm{A}_3(t+1) = \mathrm{A}_3(t) \wedge \mathrm{A}_4(t+1) = \mathrm{A}_4(t) \wedge \mathrm{A}_5(t+1) = \mathrm{A}_5(t) \\
&\wedge \mathrm{A}_6(t+1) = \mathrm{A}_6(t) \wedge \mathrm{A}_7(t+1) = \mathrm{A}_7(t) \wedge (\forall x.\mathrm{Mem}_x(t+1) = \mathrm{Mem}_x(t)) \\
&\wedge (\mathrm{N}(t+1) = \mathrm{N}(t)) \wedge (\mathrm{Z}(t+1) = \mathrm{Z}(t)) \wedge (\mathrm{V}(t+1) = \mathrm{V}(t)) \\
&\wedge (\mathrm{C}(t+1) = \mathrm{C}(t)) \wedge (\mathrm{X}(t+1) = \mathrm{X}(t)) \ ,
\end{aligned}$$

while the simplified version is

$$\mathrm{PC}(t) = 7 \wedge \mathrm{PC}(t+1) = 2 \wedge \mathrm{D}_0(t+1) = \mathrm{D}_0(t) \wedge \mathrm{D}_1(t+1) = \mathrm{D}_1(t) \ .$$

Even if this kind of abstraction is sound with respect to div-simulation, we think it is more natural and efficient to provide it as an independent tool; of course, the same simplification is operated over the ωCCS representation.

Another important kind of abstraction which is described more naturally in higher-order logic than in ωCCS is the mapping between abstract data types and their concrete representations.

In this case we use a `datatype` declaration of Isabelle [7] with an explicit instance for the representation function.

```
datatype string = null | char of (byte,string)
```

$$\begin{aligned}
\mathrm{rep}(\texttt{null})(t) &= 0 \\
\mathrm{rep}(\texttt{char}(x,s))(t) &= n \rightarrow \mathrm{Mem}_n(t) = x \wedge n \neq 0 \\
&\quad \wedge (s = \texttt{null} \vee (s \neq \texttt{null} \wedge \mathrm{rep}(s)(t) = n+1))
\end{aligned}$$

Fig. 10. Abstraction over data type: C strings.

Informally, it means that we define explicitly what is the map from the abstract data type to its concrete representation which is the one manipulated by the object code. As an example, Fig. 10 shows how a string (in the standard C representation) is coded following these guidelines. In this way we can describe by means of the data type operations what is performed by the program.

Of course, our main abstraction mechanism is div-simulation and, since the ωCCS theory is coded into higher-order logic, it is immediately available when it is necessary to perform abstraction steps which are beyond the capabilities of our other tools.

There is a simple map from ωCCS processes to logical formulas which enable us to translate the abstracted process into an abstract logical representation for the program. This map is essentially a formulation of semantics for ωCCS into higher-order logic and its definition is standard [5].

5 Summary and further work

A verification formalism based on two different frameworks — a higher-order logic and a process algebra — has been discussed in this paper. We have shown that a very general form of abstraction (simulation) can be coded into a logic using a process algebra. Since abstraction is a crucial issue in applying formal methods in practice, this method is a significant step towards a feasible object code verification of representative programs.

When emphasising the practical motivation of this paper, we have also introduced ωCCS as process algebra which is a slightly modified version of CCS. Our modifications allow a better application of this concept without sacrificing too many characteristics of CCS.

The results in this paper are direct spin-offs of an ongoing project concerned with the development and use of a system that applies formal methods to "real world" program verification. We use higher-order logic as specification language and proof environment, interfacing to the user, and ωCCS as machinery behind, to handle and modify the object of interest (in our case, programs in assembly language). So, two powerful techniques are applied to particular parts of the verification process. The fusion of these two frameworks is the key part of a successful application and has now been presented by this paper.

Moreover, the results of this work have actually been implemented and used. The ωCCS formalism has been coded within the Isabelle/HOL prover. Isabelle allows us to specify new object logics within the prover and immediately use these logics to drive new proofs. We have therefore been able to prove the correspondence between HOL and the lemmas regarding ωCCS, in particular div-simulation, within the prover. This gives us a useful framework to apply abstraction for practical purposes.

We are now in the process of using this proof infra-structure to aid in the verification of object code programs of significant size. The example suite we use is the GNU C Library compiled for the Motorola 68000 architecture.

This paper presents a general idea of "implementing" abstraction into a higher-order logic. We feel that we have developed an interesting verification methodology, which is practical and which builds on results of more than one theoretical framework. Our result encourages the application of this approach to other logics, as well.

Acknowledgements

Carl Pulley has been responsible for many of the ideas embodied in this work, and a substantial part of the implementation. Mathai Joseph proposed the project and provided much of the guidance. This work is sponsored by EPSRC under grant GR/K52447.

References

1. J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
2. Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
3. Robin Milner. An algebraic definition of simulation between programs. In *Second Joint Conference on Artificial Intelligence*, pages 481–489, 1971.
4. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
5. Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, London, 1989.
6. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer-Verlag, 1994.
7. Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. Technical Report 304, Computer Laboratory, University of Cambridge, May 1997.
8. Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, Computer Laboratory, University of Cambridge, May 1997.
9. D. Pavey and L. Winsborrow. Demonstrating equivalence of source code and PROM contents. *The Computer Journal*, 36(7):654–667, 1993.
10. David Walker. Bisimulation and divergence. *Information and Computation*, 85:202–241, 1990.
11. Yuan Yu. Automated proofs of object code for a widely used microprocessor. Research Report 114, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, October 1993.