Verification of Starlog Programs

John Cleary and Mark Utting
Department of Computer Science
The University of Waikato
Private Bag 3105, Hamilton, New Zealand
Email: {jcleary,marku}@cs.waikato.ac.nz

Abstract

Starlog is a purely declarative, temporal, logic programming language. It supports both bottom-up and top-down execution and is well-suited to writing reactive programs. This paper presents a simple methodology for proving Starlog programs correct with respect to predicate calculus specifications.

The promise of declarative programming languages is that they make it easier to write provably correct programs because the languages have simple semantics, so programs are more amenable to proof than imperative programs.

However, this promise has not been fully delivered in languages such as Prolog, because the complexities of side-effects, negation-by-failure, incomplete search strategy etc. make it necessary to reason about its operational semantics, rather than the simpler declarative (logical) semantics. Furthermore, the pure subset of Prolog is not powerful enough to write reactive programs that perform I/O.

In this paper we describe Starlog [SU99], a pure logic programming language with an explicit notion of time, and show that it has a simple and elegant proof theory which can be applied to reactive programs. The key to this simplicity is that the declarative semantics of Starlog matches its operational semantics, so it is sufficient to perform verification or refinement within the declarative framework.

This allows a simple method of proving programs correct. We model both specifications and programs simply as predicate calculus formulae [Heh91, Heh93]. An implementation I correctly implements a specification S iff:

$$I \Rightarrow S$$

is universally true.

In practice, it is often useful to write the specification as $A \Rightarrow E$, where A captures the *assumptions* about the environment of the specified system, and E expresses the *effects* of the system, such as the outputs that it must produce. Note that this gives the usual *refinement* ordering over specifications [BvW98, Mor90, Heh93]: a specification can

be refined (improved) by either weakening its assumptions (thus making it usable in more environments) or by strengthening its effects (being more precise about outputs). It is good specification style to specify just the essential or interesting properties of the outputs, so the exact outputs are typically underspecified, and strengthening the effect corresponds to reducing this underspecification.

Our correctness proof can be written equivalently as:

$$A \wedge I \Rightarrow E$$

To allow for real-time and reactive programming, our formulae model inputs, outputs and state variables as *traces* over time. That is, variables representing time appear explicitly in the clauses of the logic program. We use natural numbers to model time, with each increment representing some internal computation step. It is also possible to use real numbers for time, which is particularly useful when specifying real-time systems [HU98, MH91, Mah92].

What are the practical benefits of such a proof methodology for Starlog?

- The main benefit is that the specification gives an alternative view of the program, and the proof ensures that this view is consistent with the actual implementation. This form of redundancy is a valuable way of increasing confidence in the behaviour of the system.
- We often specify and prove a set of predicates that are intended to be used as a component of other systems, like the resource manager in Section 3. In this case, the specification defines the contract between the component and the rest of the system, clarifying both the properties of the component (E in the above formula) as well as the component's assumptions (A) about its clients. So another practical benefit is that doing the correctness proof often exposes implicit assumptions about the client's behaviour, forcing the specification to be revised by strengthening A.
- Finally, a third benefit of having a clear and simple proof methodology is that it gives us a sound basis for justifying compiler transformations and optimizations.

1 Overview of Starlog

A Starlog program comprises several timed predicates, where each predicate is a set of clauses with the following form:

```
    click(X,Y)@T,
    10 =< Y, Y =< 20,
    not paused@T
--> circle(X,Y,3)@T+1.
```

Informally, this reads: *output a circle/3 fact at time $T + 1$, whenever the mouse is clicked at time T and the Y-coordinate of the mouse click is between 10 and 20 and the* `paused` *predicate is not true*. The circle/3 fact might be an output of the whole program (perhaps displayed on a screen) or it might be an internal fact that will be used by other predicates in the program.

The execution of the clause above is bottom-up as emphasized by the fact that the body of the clause is to the left of the head, that is the execution order follows the typographical order. Starlog also permits top-down execution, and the mixing of the two styles in one clause. The following program shows an example of this.

```
    click(X,Y)@T0
--> left(Z)@T <--
        Z<X, T0<T.
```

Again clicking on the mouse will be recorded at coordinates X and Y and time T0. Later if a (top-down) call is made to `left(Z)@T` it will succeed if `Z` is to the left of the mouse. Notice that the order of execution is left to right through all parts of the clause. The semantics of `-->` and `<--` are simply logical $\Rightarrow$ and $\Leftarrow$, respectively. So the above clause, written in the usual logical syntax, is just:

$$
\begin{array}{ll}
 & click(X,Y)@T0 \wedge \\
\Rightarrow & (left(Z)@T \Leftarrow \\
 & \quad Z < X \wedge T > T0).
\end{array}
$$

Note that we use non-negative integer times, and in this paper they represent internal computation times rather than real time, so the time $T + 1$ simply means *immediately after* time T. As shown above, the body of a Starlog clause can contain negations that call other predicates, perhaps recursively. However, we require all clauses to be *stratified*, so that literals in the body must have a strictly earlier timestamps than the head of the clause. This ensures that every clause is *causal*. This stratification means that the usual two-valued fixed-point semantics is well-defined and gives the same results as the fixed-point semantics using three-valued logic [PP90].

Starlog programs are executed bottom-up, in the order given by the explicit timestamps (starting at zero). As execution proceeds, a set of timed facts is derived that describes the history of the Starlog world up to the current time. Facts from the external world, such as mouse clicks, can be inserted into this database with the current timestamp, and can therefore influence future computation. This allows reactive systems to be described in a natural style. Thanks to the explicit timestamps, it is easy to write programs that model state changes, such as assignments and changes to the external world. This natural style is one of the main attractions of the approach.

We have several prototype implementations of Starlog: one requires all generated facts to be ground and has efficient support for destructive update of state variables [SU99]; another implementation uses a constraint system to handle a wider class of programs

(where a single firing of a clause can produce an infinite set of tuples); and a third implementation does type and mode checking similar to that of Mercury [SHC96] then translates the Starlog into C or Java. The C output of this compiler can be compiled for the LEGO[1] MINDSTORMS RCX Robotics Controller, allowing simple robot control programs to be written in Starlog.

Currently, we are extending Starlog to allow more flexible stratification ordering, as well as nested timestamps. We are also working on a system that selects efficient data structures for each predicate, based on the ways that the program accesses it.

The rest of this paper shows two example programs and corresponding proofs. The next section gives a very simple example in order to demonstrate the methodology that we described above. Section 3 gives a much more substantial example and a small part of the proof. The final section summarizes the approach and outlines future directions for the research.

2 Proving 'even' correct

To re-iterate, the process of proving the correctness of a Starlog program consists of the following steps applied to the assumptions A, the program P, and the effects of the program E:

- Convert the clausal logic program P to its completion I.
- Check that the program is stratified.
- Prove the implication $A \wedge I \Rightarrow E$.

To demonstrate this process we consider the following program to compute all positive even numbers. The program starts at time 0 and enumerates the tuples `even@0`, `even@2` ..., in time order. The program P is:

```
        not even@S,
        T is S+1,
        T >= 0,
    --> even@T.
```

As described earlier, the correctness of the program is proven by showing that $A, I \Rightarrow E$. This example is very simple and the assumption A is empty. The completion of the program (I) is:

$$I \equiv (\forall T \bullet even@T \Leftrightarrow (\exists S \bullet T \geq 0 \wedge T = S + 1 \wedge \neg(even@S)))$$

[1] LEGO, MINDSTORMS, Robotics Invention System and RCX are trademarks of the LEGO Group. See `www.legomindstorms.com` for more details.

and the effect E is

$$E \quad \equiv \quad (\forall T \bullet even@T \Leftrightarrow T \geq 0 \wedge (\exists N \bullet T = 2 * N))$$

Now we show that the program is stratified. We do this by defining a strict partial order $\ll$ over the ground terms. In this case the assertion that $even@T \ll even@T + 1$ is sufficient. Clearly $\ll$ is a partial order over all terms $even@T$ where T is an integer and the head of the clause is always greater than the body literals.

Next we show the program is correct by proving $I \Leftrightarrow E$ (this is stronger than necessary—$I \Rightarrow E$ would suffice).

The proof is by induction on the time T. That is, I and E are restricted to $T \leq t$ and then it is shown that the implication holds for $t + 1$. First, note that the result holds for $T = 0$ and $T = 1$. For $T = 0$, the completion of the program implies that $even@T$ is false for $T < 0$ and so by definition $even@0$ is true. E also implies that $even@0$ is true, since $0 \geq 0 \wedge 0 = 2 * 0$ is true. For $T = 1$, I implies that $even@1$ is false (as $1 = 0 + 1 \wedge \neg(even@0)$ is false) and E also implies that $even@1$ is false (there is no N such that $1 = 2 * N$).

For the inductive step, we assume that the right-hand-sides of I and E are equivalent at time T (for any $T \geq 0$), then show that they are equivalent at time $T + 1$. Note that all quantifiers range over natural numbers only.

$$\begin{array}{ll}
& (\exists S \bullet T + 1 \geq 0 \wedge T + 1 = S + 1 \wedge \neg(even@S)) \\
\Leftrightarrow & \{\text{One-point rule, since } T + 1 = S + 1 \text{ means that } T = S\} \\
& T + 1 \geq 0 \wedge \neg(even@T) \\
\Leftrightarrow & \{\text{Applying the inductive hypothesis to } even@T\} \\
& T + 1 \geq 0 \wedge \neg(T \geq 0 \wedge (\exists N \bullet T = 2 * N)) \\
\Leftrightarrow & \{\text{Since } T \geq 0 \text{ is true}\} \\
& T + 1 \geq 0 \wedge (\forall N \bullet T \neq 2 * N) \\
\Leftrightarrow & \{\text{By mathematical properties of evenness}\} \\
& T + 1 \geq 0 \wedge (\exists N \bullet T + 1 = 2 * N)
\end{array}$$

3 Resource Manager

3.1 Program and Completion (P, I)

The program in Fig. 1 implements a system which grants a sychronised token to a requester. It guarantees that only one requester at a time can be given a token. The algorithm is fair - a request is always eventually granted provided the resource is always eventually freed after being granted. In this particular implementation, requests are

granted in the order that they are made. If two requesters make a request at the same time, then the requester with the smaller identifier will be granted the resource first.

The external interface of this program comprises these tuples:

request(X)@T: a request is made by requester X at time T. The resource identifier, X, must be an integer in this implementation.

grant(X)@T: the program grants the resource to requester X at time T.

free(X)@T: the resource is freed again by the requester X at time T. It is the requestor's responsibility to ensure that a free is always generated after a grant.

Internally, the program uses several state variables, which are accessed by `is` and are set and cleared by `initiates` and `terminates` facts. Their implementation uses **`delete`** facts.

free: the resource is currently free.

requested(T0,X): requester X made a request at time T0, and that request is still unsatisfied.

Other internal predicates are:

free: the resource has been freed by some requester (also used to initialize the free token at time 0).

grant(T0,X): The token is granted to X in response to a request at time T0.

req_fifo(T0,X): There is another unsatisfied request for the token, which is earlier than the one for X at time T0.

lt(_,_): This is a top-down predicate which is used to order time/identifier pairs.

3.2 Specification (E)

The specification of the resource manager is not a complete one. Unlike the even program it is not possible or advisable to predetermine all of the programs behaviour.

The first few requirements are concerned with the situation when a $grant$ is issued. The first says that every $grant$ has a preceding request. That is, a grant cannot be issued spontaneously.

$$grant(T, X)@T' \Rightarrow T' > T, request(X)@T$$

This proposition is in terms of the two-valued form of the grant which refers to the time that the corresponding request was made. Strictly speaking the two-valued form

```
free@0.   % Make sure resource is free at the beginning

free@T --> initiate(free)@T.

free(X)@T --> free@T.

grant(_,_)@T --> terminate(free)@T.

request(X)@T0 --> initiate(requested(T0,X))@T0.

grant(T0,X)@T --> terminate(requested(T0,X))@T.

% Is there another request current earlier than T0
% (tie break on identifier if at same time)
    is(requested(T1,Y))@T
--> req_fifo(T0,X)@T <--
        lt(requested(T1,Y),requested(T0,X)).

lt(requested(T1,Y), requested(T0,X)) <-- T1<T0.
lt(requested(T,Y), requested(T,X)) <-- Y<X.

% The heart of the logic:
%  if a request and resource is free and there is no
%  earlier unsatisfied request then grant the resource
    is(free)@T,
    is(requested(T0,X))@T,
    not(req_fifo(T0,X)@T)
--> grant(T0,X)@T.

    grant(T0,X)@T
--> grant(X)@T.

%Utility routines for initiates/terminates/is
    initiate(F)@T0,
    T>T0
--> is(F)@T <--
        not(delete(F,T0)@T).

delete(F,T0)@T <-- terminate(F)@T1, T0<T1, T1<T.
```

Figure 1: Resource Manager Program in Starlog

is internal to the implementation, however, giving it in this form is more precise. The completion of the program includes: $grant(X)@T' \Leftrightarrow \exists T \bullet grant(T,X)@T'$ so the requirement above can be written in the weaker form:

$$grant(X)@T' \Rightarrow (\exists T \bullet T' > T, request(X)@T)$$

The next requirement says that if a $grant$ is given then there cannot have been an earlier $grant$ for the same request:

$$grant(T,X)@T' \Rightarrow \neg(\exists S \bullet grant(T,X)@S, S < T')$$

The third requirement is that only one $grant$ can be given at a time:

$$grant(X)@T, grant(Y)@T \Rightarrow X = Y$$

The next requirements specify safety conditions for when a $grant$ can be given. To help in this an auxiliary predicate $taken$ is specified. $taken(X)$ is true so long as there is an outstanding $grant$ for X which has not yet been freed. The form $taken$ is true if any $grant$ is currently outstanding.

$$\begin{aligned} taken(X)@T \equiv\ & (\exists T' \bullet grant(X)@T' \wedge T' < T \wedge \\ & \neg(\exists T'' \bullet \mathit{free}@T'' \wedge T' \leq T'' \leq T)) \end{aligned}$$

$$taken@T \equiv (\exists X \bullet taken(X)@T)$$

The first requirement using these definitions is that a $grant$ can only be issued if it is not currently taken:

$$grant(X)@T \Rightarrow \neg taken@T$$

The second is that only one $grant$ can be outstanding at a time:

$$taken(X)@T \wedge taken(Y)@T \Rightarrow X = Y$$

The strongest of the requirements on the program is that it should eventually give a $grant$ to every request:

$$request(X)@T \Rightarrow (\exists T' \bullet grant(X,T)@T' \wedge T' \geq T)$$

This requirement is notable as it is a fairness condition and highlights the ease with which such conditions can be stated when the time values are made explicit in the code.

The following two conditions might be considered optional. They both specify the order in which requests will be granted. They are provably true of this particular implementation but might easily be omitted for a general specification which does not bind the programmer too strongly.

The first of these optional requirements says that requests are granted in the order of the time at which they were made:

$$grant(T', X)@T, grant(U', Y)@U, T' < U' \Rightarrow T < U$$

The second is stronger (and more optional) and says that requests at the same time are granted in identifier order:

$$grant(T', X)@T, grant(T', Y)@U, X < Y \Rightarrow T < U$$

3.3 Assumptions (A)

As well as the requirements on the implementation itself there are also assumptions about the code which uses the request program. The first of these is a simple type restriction that $request$s and *free*s have integer identifiers and positive integer time values:

$$request(X)@T \Rightarrow integer(X), T \geq 0$$

$$\mathit{free}(X)@T \Rightarrow integer(X), T \geq 0$$

The next assumption is that there can only be a finite number of requests at one time. This assumption is a subtle one that would not normally arise in other contexts (for example in an imperative language) as it would be implicit in the computational model being used. Another way of viewing the assumption is that it forbids a livelock where a $grant$ is postponed indefinitely. The assumption is also essential for proving the specification that all requests are eventually granted.

$$(\exists N M \bullet (\forall X \bullet request(X)@T \Rightarrow M \leq X \leq N))$$

The next assumption is that every $grant$ is eventually *free*d. Again this is a assumption that is essential to proving that every request is eventually granted.

$$grant(X)@T \Rightarrow (\exists T' \bullet T' > T \land \mathit{free}(X)@T')$$

The next two assumptions are that every *free* has a corresponding earlier $grant$ and that there is at most one *free* for each $grant$. This assumption is a dual to the requirements on $grant$ in the program specifications.

$$\mathit{free}(X)@T \Rightarrow (\exists S \bullet S < T \land grant(X)@S)$$

$$\mathit{free}(X)@T \Rightarrow \neg(\exists S \bullet S < T \land \mathit{free}(X)@S)$$

3.4 Stratification

We now show that the program is stratified by exhibiting a well founded partial order on the ground terms of the program. The well-foundedness is shown in most cases by that fact that all the time values are non-negative. It is also necessary to show that each clause in the program respects the partial ordering, that is, that the head of the clause is always greater than the literals in the body. The partial ordering is specified as a logic program - we are currently experimenting with this style of ordering specification for user declared and compiler inferred ordering relationships.

We give two different partial orderings. The first is simpler and can be easily hand checked using a diagram. The second is more precise and gives more freedom to a compiler in choosing the order to execute code which might as a result be more efficient.

The first part of the ordering says that all terms are ordered on their time values:

$$A@S \ll B@T \Leftarrow T < S$$

and that if they have the same time then the ordering is on the full predicate:

$$A@T \ll B@T \Leftarrow A \ll B$$

The code below gives the ordering between symbols at the same time:

$$\mathit{free}(_) \ll \mathit{free} \ll initiate(_)$$

$$request(_) \ll initiate(_)$$

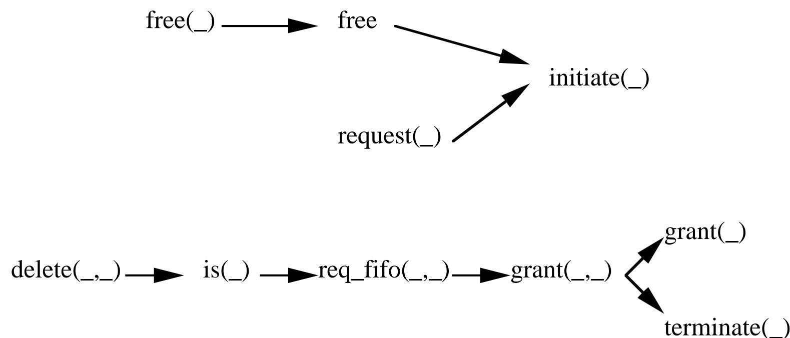
$$delete(_,_) \ll is(_) \ll \mathit{req_fifo}(_,_) \ll grant(_,_) \ll grant(_)$$

$$grant(_,_) \ll terminate(_)$$

The predicate $lt(_,_)$ is like a built-in predicate and is ordered before any time value:

$$lt(_,_) \ll \mathit{req_fifo}(_,_)@T$$

The following figure verifies that this partial ordering has no cycles.



The partial ordering above is more restrictive than is strictly necessary. The version below is weaker and gives more freedom to the run time system or to a compiler to vary the order of execution.

Most of the ordering is between tuples at the same time (the constraint that *all* tuples are ordered on time is dropped in this version).

$$A@T \ll B@T \Leftarrow A \ll B$$

However, the code using the predicates is, $initiate$, $terminate$ and $delete$ which implements state variables does require some inter-time ordering.

$$initiate(F)@S \ll is(F)@T \Leftarrow S < T$$

$$terminate(F)@S \ll delete(F, _)@T \Leftarrow S < T$$

Note that the orderings are indexed on the state variable names so these rules do not imply any ordering between the different state variables.

As above lt is ordered before any *req_fifo* value:

$$lt(_, _) \ll \mathit{req_fifo}(_, _)@T$$

The remainder of the ordering is between values at the same time:

$$\begin{array}{l} \mathit{free}(_) \ll \mathit{free} \ll initiate(\mathit{free}) \\ request(X) \ll initiate(requested(_, X)) \\ is(requested(_, _)) \ll \mathit{req_fifo}(_, _) \\ is(requested(T, X)) \ll grant(T, X)) \\ \mathit{req_fifo}(T, X) \ll grant(T, X) \\ is(\mathit{free}) \ll grant(_, _) \\ grant(_, X) \ll grant(X) \\ delete(F, _) \ll is(F) \end{array}$$

3.5 Correctness Proof

There are some seven different proofs that are required from the specification of the program - one for each of the requirements. Space precludes us from giving all the proofs. However, two of them are given below as good illustrations of the type of proof that results from our approach.

The first of these shows that $request$s are granted in time order.

Theorem 1 $\quad grant(T', X)@T \wedge grant(U', Y)@U \wedge T' > U' \Rightarrow T > U$

Proof: Most of the steps in the proof are the substitution of terms by equivalent terms using the completion of the program. Such transformations will be done without comment in the proof.

The proof as a whole assumes $T \leq U$ and shows a contradiction.

$$
\begin{array}{ll}
\multicolumn{2}{l}{grant(T', X)@T \wedge grant(U', Y)@U \wedge T' > U'} \\
\Leftrightarrow & \{\text{By the completion of } grant(T', X)@T\} \\
& T \leq U \wedge U' < T' \wedge \\
& is(\mathit{free})@T \wedge is(requested(T', X))@T \wedge \neg \mathit{req_fifo}(T', X)@T \wedge \\
& grant(U', Y)@U
\end{array}
$$

Note that $U' < T' \leq T \leq U$. It will be shown that *req_fifo*$(T', X))@T$ succeeds thus leading to the contradiction.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{req_fifo}(T', X)@T} \\
\Leftrightarrow & (\exists T_1, Y' \bullet is(requested(T_1, Y'))@T \wedge \\
& \qquad lt(requested(T_1, Y') \wedge requested(T', X))) \\
\Leftarrow & \{\text{replacing } T_1 \text{ by } U' \text{ and } Y' \text{ by } Y\} \\
& is(requested(U', Y))@T \wedge lt(requested(U', Y) \wedge requested(T', X)) \\
\Leftrightarrow & \{\text{Since } U' < T' \text{ then } lt(requested(U', Y), requested(T', X)) \text{ is true}\} \\
& \mathit{req_fifo}(T', X)@T \\
\Leftrightarrow & is(requested(U', Y))@T \\
\Leftrightarrow & (\exists T_0 \bullet initiate(requested(U', Y))@T_0 \wedge \\
& \qquad T > T_0 \wedge \neg delete(requested(U', Y), T_0)@T) \\
\Leftarrow & \{\text{Replacing } T_0 \text{ by } U'\} \\
& initiate(requested(U', Y))@U' \wedge \\
& \qquad T > U' \wedge \neg delete(requested(U', Y), U')@T) \\
\Leftrightarrow & \{\text{By completion of } initiate\} \\
& request(Y)@U' \wedge T > U' \wedge \neg delete(requested(U', Y), U')@T) \\
\Leftrightarrow & \{request(Y)@U' \text{ follows from the assumption } grant(U', Y)@U \text{ by}\} \\
& \{\text{the effect properties. Also } T > U' \text{ is true by assumption.}\} \\
& \neg delete(requested(U', Y), U')@T) \\
\Leftrightarrow & \neg(\exists T_1 \bullet terminate(requested(U', Y))@T_1 \wedge U' < T_1 \wedge T_1 < T) \\
\Leftarrow & \neg(\exists T_1 \bullet terminate(requested(U', Y))@T_1 \wedge T_1 < T) \\
\Leftarrow & \{\text{From the effect properties}\} \\
& grant(U', Y)@U
\end{array}
$$

That is, $grant(U', Y)@U \Rightarrow$ *req_fifo*$(T', X))@T$, which shows that the initial equivalence fails, which gives the contradiction that we sought. □

The second proof we will give is a sketch of the proof that every $request$ is eventually granted.

Theorem 2 $\quad request(X)@T \Rightarrow (\exists T' \bullet grant(X, T)@T' \wedge T' \geq T)$

Proof: The proof proceeds by induction on the $request$ terms. They are considered in the lexicographic ordering of time and requester identifier. The first request needs to be treated specially as the initial *free* is automatic not generated by a $grant$. Then assuming that all $request$s up to some pair T, X have been granted and assuming that they have all been eventually *free*d there will be a time T'' which will be the time of the last *free* for all the $grant$s corresponding to the requests. A little more effort shows that this free will be that corresponding to $request(X)@T$ but this is not essential.

Now it is necessary to show that the next $request$ after T, X, say T', X' will eventually be granted. There are two cases to be considered depending on whether $T' < T''$ or $T' \geq T''$.

If $T' < T''$ then it can be shown that $grant(T', X)@T'' + 1$ will be generated by the program. That is, the $grant$ will be issued immediately following the final *free*. Similarly, if $T' \geq T''$ then $grant(T', X)@T' + 1$ will be generated. That is, the $grant$ will be issued immediately following the request.

□

4 Conclusions

We have described a simple proof methodology for a pure logic programming language that supports reactive programming. The resource manager example illustrates how this can be used on a component of a larger reactive system.

The method is simple, basically just proving an implication in the standard predicate calculus. But of course, the proofs themselves can be complex, depending upon the complexity of the Starlog program, and upon how large a gap there is between the abstraction level of the specification and that of the program. A key point is that the program is data-structure free, which means that inductive proofs over data structures are not required.

The observation that many kinds of verification or refinement can be done purely within the standard predicate calculus has been eloquently espoused by Hehner [Heh91, Heh93]. He shows that it leads to very simple calculi. However, he deals with imperative languages, so must still embed program language constructs within predicate calculus, then derive laws for manipulating those constructs.

In contrast, we have worked just with the standard predicate calculus, because our programming language is declarative and close enough to pure logic that it can execute

fragments of predicate calculus. The translation from that form into a Starlog program is almost mechanical (replacing $\Leftrightarrow$ with `-->`, and flattening clauses etc).

The key to the simplicity is that the target programming language is a pure declarative language, and the compiler performs sufficient checking to ensure the following property:

all programs accepted by the compiler have an operational semantics that matches their declarative semantics.

This allows all proving or refinement to be done purely in the declarative world, and avoids the complexities of a proof system that models the operational semantics of Starlog.

To satisfy this property, compilers must use techniques like strong type checking, mode-checking (to check or reorder calls so that arguments are instantiated before predicates are called), determinism checking, stratification checking and/or termination checking. One of our current Starlog compilers (Starlog/NQC) performs type checking, stratification checking (which guarantees progress) as well as simple mode checking to eliminate instantiation errors. We are currently extending these techniques to a larger subset of Starlog.

Standard Prolog does not have the above property, unless we restrict our programming to its pure subset, and this is not expressive enough to describe reactive programs or any form of I/O. The best-known pure logic programming language, Mercury [SHC96], is capable of reactive programming, while remaining pure, because its mode system ensures that the state of the external world is single-threaded through the program. This is similar to the use of monads in functional programming [Wad95]. The Starlog approach is rather different, and perhaps more elegant, because one reasons directly about time, and changes over time, rather than encoding this indirectly via monads or some other single threading mechanism.

Acknowledgements

Thanks to the other members of the Starlog team, especially Bernhard Pfahringer and Roger Clayton for many insightful discussions. This research is partly funded by the Marsden project *Temporal Declarative Programming*, (UOW605).

References

[BvW98] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Approach*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Heh91] E.C.R. Hehner. *Advances in Computing and Information*, volume 497 of *Lecture Notes in Computer Science*, chapter What's wrong with formal programming methods?, pages 2–23. Springer, 1991.

[Heh93] E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.

[HU98] I. J. Hayes and M. Utting. Deadlines are termination. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods: PROCOMET '98, IFIP TC2/WG2.2, 2.3 International Conference, June 1998, Shelter Island, New York*, pages 186–204. Chapman and Hall, 1998.

[Mah92] Brendan Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, University of Queensland, February 1992.

[MH91] B. P. Mahony and I. J. Hayes. A case study in real-time specification: A central heater. In Joseph M. Morris and Roger C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, pages 138–149. Springer-Verlag, January 1991. Workshops in Computing Series.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[PP90] H. Przymusinska and Przymusinski. Weakly stratified logic programs. *Fundamenta Informatica*, 13:51–65, 1990.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, October-December 1996.

[SU99] Donald A. Smith and Mark Utting. Pseudo-naive evaluation. In *Tenth Australasian Database Conference, ADC'99, Auckland, New Zealand, Jan 1999*. Springer-Verlag, 1999.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Baastad, Sweden*, number 925 in LNCS, pages 24–52. Springer-Verlag, 1995.