Rewriting UNITY

Adam Granicz, Daniel M. Zimmerman, and Jason Hickey

California Institute of Technology
Computer Science Dept. MC 256-80
Pasadena CA 91125, USA
{granicz,dmz,jyh}@cs.caltech.edu

Abstract. In this paper we describe the implementation of the UNITY formalism as an extension of general-purpose languages and show its translation to C abstract syntax using Phobos, our generic front-end in the Mojave compiler. Phobos uses term rewriting to define the syntax and semantics of programming languages, and automates their translation to an internal compiler representation. Furthermore, it provides access to formal reasoning capabilities using the integrated MetaPRL theorem prover, through which advanced optimizations and transformations can be implemented or formal proofs derived.

1 Introduction

UNITY [3] is a powerful formalism for the specification of nondeterministic concurrent programs. The UNITY language and execution model are simple, yet there has been little effort directed toward the compilation of UNITY programs to executable code. In this paper we present a method that uses Phobos [7], the generic front-end of the Mojave [12] compiler, to translate UNITY programs into C abstract syntax suitable for code generation. Our method has concrete advantages over previously known techniques for generating executable code from UNITY programs: the implementation is quickly adaptable to different target languages, we can easily change the scheduling algorithm used in the generated code, and we can leverage the attached theorem prover to carry out transformations and proof derivations.

In our implementation we eliminate nondeterminism from UNITY programs by using a simple sequential scheduling of statements, which may consist of simple, conditional or quantified assignments as defined in the formalism. This particular scheduling is not an inherent property of the translation method, and can be easily modified as we describe later. The entire implementation is small, and can be tailored to different target languages with minimal effort. We do not address formal properties in this paper, but the implementation is designed to lay the groundwork for formal analysis in the MetaPRL system.

1.1 Related Work

Few compilers have been developed for the UNITY language. DeRoure's parallel implementation of UNITY [5] compiles UNITY to a common backend language, BSP-occam; Huber's MasPar UNITY [11] compiles UNITY to MPL for execution on MasPar SIMD computers; and Radha and Muthukrishnan have developed a portable implementation of UNITY for Von Neumann machines [17]. These UNITY compilers are not as easily adaptable to multiple target languages or multiple scheduling algorithms as the rewriting-based translator we describe, and none of them have the formal reasoning capabilities provided by an integrated theorem prover.

The construction of formal proofs for UNITY programs has been mechanized using various theorem proving environments. Anderson's HOL-UNITY [2] is an implementation of UNITY using the HOL system [6], Heyd and Cregut's Coq-UNITY [8] uses Coq, and Paulson has implemented UNITY within the Isabelle environment [15, 16]. While these implementations provide assistance in proof generation for UNITY programs, they do not generate executable code.

2 The UNITY Formalism

The UNITY formalism consists of both a programming language (with accompanying execution model) and a proof logic. In this paper we are primarily concerned with the language and its execution model. For a discussion of the proof logic the reader is referred to Chandy [3].

2.1 Language

In the UNITY programming language, a program begins with a **program** declaration that specifies the program's name. This is followed by several program sections:

1. A **declare** section, which names the program variables and declares their types.
2. An optional **always** section, which defines program variables as functions of other variables. Variables defined in this section are essentially textual macros representing these functions, rather than actual state variables of the program.
3. An **initially** section, which specifies initial values for the variables from the **declare** section. Uninitialized variables have arbitrary initial values.
4. An **assign** section, the program's body, which consists of a set of assignment statements. These statements may be single or multiple assignments, and may be conditional through the use of an **if** construct. They may also be quantified over predetermined ranges using the $[\!]$ operator, which represents nondeterministic choice. In a multiple assignment statement, all the expressions on the right side and any subscripts on the left side are evaluated first, then the values of the expressions on the right side are assigned to the corresponding variables on the left side.

An example UNITY program that sorts an array of N elements is shown in Figure 1. The initialization sets the values for the elements of the array in parallel, and the quantified assignment is a nondeterministic choice among $N-1$ multiple assignment statements. In our implementation we do not deal with parallelism, but instead replace it with nondeterminism.

```
program  array-sort

    declare
        a: array [N] of integer

    initially
        ⟨|| i : 0 ≤ i < N :: a[i] = N − i⟩

    assign
        ⟨[] i : 0 ≤ i < N − 1 :: a[i], a[i + 1] := a[i + 1], a[i]   if   a[i] > a[i + 1]⟩

end
```

Fig. 1. A UNITY program that sorts an array of N integers

2.2 Execution model

Execution of a UNITY program proceeds in the following way. First, the initialization statements are executed, simultaneously, to set the state variables to their initial values. Then, statements are repeatedly selected and executed atomically. Statement selection is subject to a weak fairness constraint, which requires that every statement is selected infinitely often in every infinite execution of the program. There are no other constraints on statement selection, so some statements may be executed far more often than others in any finite execution prefix.

It is possible for a UNITY program to reach a *fixed point*, where there is no statement whose execution would change the value of any state variable. When this occurs, we say that the program has terminated.

3 System Architecture

The Mojave compiler supports various front-end languages which are translated to a common functional intermediate representation. The typical code path through these source languages is shown in Figure 2 as (I). In addition, the integrated MetaPRL theorem prover can be used to perform transformation and formal reasoning of the programs under compilation. Phobos acts as a bridge between source languages and the formal system by providing generic parsing and transformation capabilities using the term rewriting mechanism of MetaPRL. Programming language syntax can be specified with context-free

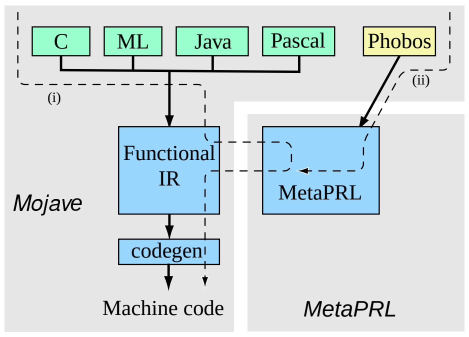

Fig. 2. The Mojave compiler architecture

grammars where rewrite rules are used to describe parser actions, and the program is represented as a term in the formal system. Program transformations and domain-specific knowledge can be specified using *formal* (which avoid capture and are guaranteed not to change binding) and *informal* (which are used for parsing and can create binding) rewrite rules that are passed to MetaPRL for execution. The final term is then converted to a specified compiler representation (in Figure 2 this is the functional IR) and compilation proceeds to generate executable code.

3.1 Term language

The term rewriting engine we use belongs to the MetaPRL logical framework [10, 14]. All logical terms, including goals and subgoals, are expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has 1) an operator-name, which is a unique name identifying the term; 2) a list of parameters representing constant values; and 3) a list of subterms with possible variable binding occurrences. We use the following syntax to describe terms, based on the NuPRL definition [1]:

$$\underbrace{opname}_{operator\ name}\ \underbrace{[p_1; \cdots ; p_n]}_{parameters}\ \underbrace{\{\boldsymbol{v_1}.t_1; \cdots ; \boldsymbol{v_m}.t_m\}}_{subterms}$$

Here are a few examples:

Shorthand	Term
1	`natural_number["1"]{}`
$\lambda x.b$	`lambda[]{x. b}`
$f(a)$	`apply[]{f; a}`
v	`variable["v"]{}`
$x + y$	`sum[]{x; y}`

Variables are terms with a string parameter giving their names; numbers have an integer parameter with their value. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

The rewriting engine used in MetaPRL is described in Hickey [9]. Rewriting rules are specified as a pair of terms $t_1 \longleftrightarrow t_2$ using second-order substitution. The term t_1, called the *redex*, contains second-order variables of the form $v[v_1; \cdots; v_n]$; and the term t_2, called the *contractum*, contains corresponding second-order substitutions of the form $v[t'_1; \cdots; t'_n]$, specifying the simultaneous substitution $v[t'_1, \ldots t'_n / v_1, \ldots, v_n]$. The following table lists a few examples:

Rewrite	$\mathsf{apply}\{\mathsf{lambda}\{v.b[v]\}; e\}\} \longleftrightarrow b[e]$
Shorthand	$(\lambda v.b[v])\ e \longleftrightarrow b[e]$
Example	$(\lambda x.x + x)\ 1 \longrightarrow 1 + 1$
Rewrite	$\mathsf{match}\{\mathsf{pair}\{u; v\}; x, y.b[x, y]\} \longleftrightarrow b[u, v]$
Shorthand	$(\mathbf{match}(u, v)\ \mathbf{with}\ x, y \to b[x, y]) \longleftrightarrow b[u, v]$
Example	$(\mathbf{match}(1, 2)\ \mathbf{with}\ x, y \to x + y) \longrightarrow (1 + 2)$

4 Implementation

We now describe the conversion of terms representing UNITY abstract syntax to C abstract syntax using source notation. The underlying actual term representation can be recovered in a straightforward manner. We also use the meta-syntax (::) to denote element insertion into a list, as used in OCaml. Occasionally, we show actual terms, in which case their names are underlined to distinguish them from abstract ones.

$\mathit{op} ::=$	$+ \mid - \mid * \mid / \mid \textbf{and} \mid \ldots \mid = \mid \neq \mid < \mid \leq \mid > \mid \geq$	binary operators
$r ::=$	$e\ \mathit{op}\ v\ \mathit{op}\ e$	range
$\mid$	$e\ \mathit{op}\ v\ \mathit{op}\ e\ \&\ e$	range with condition
$e ::=$	$i \mid f \mid \textbf{true} \mid \textbf{false}$	numbers and Booleans
$\mid$	v	variables
$\mid$	$e\ \mathit{op}\ e$	binary operation
$\mid$	$e[e]$	subscripting
$\mid$	$e(e, \ldots, e)$	function application
$\mathit{assign} ::=$	$e, \ldots, e = e, \ldots, e$	Simple assignment
$\mid$	$e, \ldots, e = e, \ldots, e\ \textbf{if}\ e$	Conditional assignment
$\mid$	$\langle \| v, \ldots, v : r, \ldots, r :: \mathit{assign} \rangle$	Quantified assignment

Fig. 3. UNITY assignment grammar

4.1 The C term set

The C term set is a straightforward implementation of the Mojave C abstract syntax type. Each OCaml constructor name is defined as a term; for instance

`C_declare of var * ty` is represented as $\underline{c\_declare}\{var; ty\}$, or `C_if of cond * true * false` is represented as $\underline{c\_if}\{cond; true; false\}$. When we can, we use C source syntax to denote these terms, for instance **if** (*cond*) *true* **else** *false*.

4.2 Design

The syntax we adopt for assignment statements in our implementation is defined in Figure 3. We achieve the independence of individual assignments in the same statement by keeping two sets of state variables. For each variable declared in the program we introduce an "alias," which stores the value of the aliased variable before entering a new statement. We use the alias for reading and the regular variable for writing, preserving the semantics of multiple assignment statements. The value of an alias may be different *only* when executing the statement that contains its aliased variable; upon exiting the statement, the two variables are synchronized. Throughout our discussion, we define the alias of variable v as $\mathbf{ALIAS}[v]$, and that of UNITY expressions as shown below.

$$\begin{array}{ll}
\mathbf{ALIAS}[i \mid f \mid \mathbf{true} \mid \mathbf{false}] \rightarrow i \mid f \mid \mathbf{true} \mid \mathbf{false} & \mathbf{ALIAS}[v] \rightarrow v\_alias \\
\mathbf{ALIAS}[e_1\ op\ e_2] \quad \rightarrow \mathbf{ALIAS}[e_1]\ op\ \mathbf{ALIAS}[e_2] & \\
\mathbf{ALIAS}[e_1[e_2]] \quad \rightarrow \mathbf{ALIAS}[e_1][\mathbf{ALIAS}[e_2]] & \\
\mathbf{ALIAS}[e_f(e_1, \ldots, e_n)] \rightarrow \mathbf{LV}[e_f](\mathbf{ALIAS}[e_1], \ldots, \mathbf{ALIAS}[e_n]) &
\end{array}$$

The lvalue expression $\mathbf{LV}[\ldots]$ has all but the outmost variable replaced with aliases, since we want to use regular variables for writing but aliases for reading. For instance, $\mathbf{LV}$`[a[i]]` $\rightarrow$ `a[i_alias]`.

$$\begin{array}{ll}
\mathbf{LV}[i \mid f \mid \mathbf{true} \mid \mathbf{false}] \rightarrow i \mid f \mid \mathbf{true} \mid \mathbf{false} & \mathbf{LV}[v] \rightarrow v \\
\mathbf{LV}[e_1\ op\ e_2] \quad \rightarrow \mathbf{LV}[e_1]\ op\ \mathbf{LV}[e_2] & \\
\mathbf{LV}[e_1[e_2]] \quad \rightarrow \mathbf{LV}[e_1][\mathbf{ALIAS}[e_2]] & \\
\mathbf{LV}[e_f(e_1, \ldots, e_n)] \rightarrow \mathbf{LV}[e_f](\mathbf{ALIAS}[e_1], \ldots, \mathbf{ALIAS}[e_n]) &
\end{array}$$

We track any change in the global state by monitoring each assignment through **SYNCHRONIZE**. We update the alias for variable v with v itself if the two are different, in which case we set the global **CHANGED** variable to true. This allows us to identify changes to the state variables.

$$\begin{array}{l}
\mathbf{SYNCHRONIZE}[v :: vars] \rightarrow \mathbf{if}\ (\mathbf{LV}[v] \neq \mathbf{ALIAS}[v])\ \{ \\
\qquad\qquad\qquad\qquad \mathbf{CHANGED} = \mathbf{true}; \\
\qquad\qquad\qquad\qquad \mathbf{ALIAS}[v] = \mathbf{LV}[v]; \\
\qquad\qquad\qquad \} :: \mathbf{SYNCHRONIZE}[vars]
\end{array}$$

The following illustrates how a simple statement with two conditional assignments is translated.

$$\texttt{x,y := y,x if x>y} \rightarrow \begin{array}{l}
\mathbf{if}\ (\mathbf{ALIAS}[x] > \mathbf{ALIAS}[y])\ \{ \\
\quad \mathbf{LV}[x] = \mathbf{ALIAS}[y]; \\
\quad \mathbf{LV}[y] = \mathbf{ALIAS}[x]; \\
\quad \mathbf{SYNCHRONIZE}[x]; \\
\quad \mathbf{SYNCHRONIZE}[y]; \\
\}
\end{array}$$

4.3 Translation

After parsing, the original UNITY program is represented as a *program* term (which we have pretty-printed in the rule below) whose subterms correspond to the declarations, identities, initializations and assignments in the program. The main step required to translate this term into C abstract syntax terms can be expressed as:

$$\begin{array}{lcl}
 & & \mathit{int}\ \mathit{main}(\ldots)\ \{ \\
\textbf{program}\ id & & \quad \mathit{int}\ \textbf{CHANGED}; \\
\quad \textbf{declare}\ v : ty, \ldots & & \quad \mathbf{C}_1[v : ty, \ldots] \\
\quad \textbf{always}\ v = e, \ldots & \rightarrow & \quad \mathbf{C}_2[inits] \\
\quad \textbf{initially}\ inits & & \quad \textbf{while}\ (!\,\textbf{CHANGED})\ \{ \\
\quad \textbf{assign}\ assigns & & \quad\quad \textbf{CHANGED} = false; \\
\textbf{end} & & \quad\quad \textbf{SUBST}[v = e, \ldots; \mathbf{C}_3[assigns]]\ \} \\
 & & \}
\end{array}$$

where $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{C}_3$ denote the translation process for declarations, initializations and assignments, respectively, as defined below:

$$\begin{array}{rl}
\mathbf{C}_1[(v : ty) :: rest] & \rightarrow \underline{c\_declare}\{var; ty\} :: \mathbf{C}_1[rest] \\
\mathbf{C}_1[\textbf{nil}] & \rightarrow \textbf{nil}
\end{array}$$

$$\begin{array}{rl}
\mathbf{C}_2[init :: rest] & \rightarrow \textbf{ASSIGN}[init] :: \mathbf{C}_2[rest] \\
\mathbf{C}_2[\textbf{nil}] & \rightarrow \textbf{nil}
\end{array}$$

$$\begin{array}{rl}
\mathbf{C}_3[assign :: rest] & \rightarrow \textbf{ASSIGN}[assign] :: \mathbf{C}_3[rest] \\
\mathbf{C}_3[\textbf{nil}] & \rightarrow \textbf{nil}
\end{array}$$

Note that we use the same translation for initializations and assignments to simplify our discussion. In the actual implementation, we have omitted the code that tracks state changes from the initializations.

Identities Given a list of variables and their identity expressions as defined in the **always** section of the source program, we simply substitute each expression in place of the variable.

$$\begin{array}{rl}
\textbf{SUBST}[(v = exp) :: rest; prog[v]] & \rightarrow \textbf{SUBST}[rest; prog[exp]] \\
\textbf{SUBST}[\textbf{nil}; prog] & \rightarrow prog
\end{array}$$

Assignments The heart of our implementation is the translation of assignment statements. Simple, conditional and quantified assignments are translated by $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, respectively.

$$\textbf{ASSIGN}[lvalues = values] \rightarrow \mathbf{A}_1[lvalues = values; lvalues]$$

$$\textbf{ASSIGN}[lvalues = values\ \textbf{if}\ cond] \rightarrow \mathbf{A}_2[lvalues = values\ \textbf{if}\ cond; lvalues]$$

$$\textbf{ASSIGN}[\langle[\!]\,cvars : quants :: assign\rangle] \rightarrow \mathbf{A}_3[cvars; quants; assign]$$

Simple assignments are translated directly, followed by the synchronization of all left-hand side expressions:

$$\begin{gathered}\mathbf{A}_1[(v = exp) :: rest; lvalues] \rightarrow \\ (\mathbf{LV}[v] = \mathbf{ALIAS}[exp]) :: \mathbf{A}_1[rest; lvalues]\end{gathered}$$

$$\mathbf{A}_1[\mathbf{nil}; lvalues] \rightarrow \mathbf{SYNCHRONIZE}[lvalues]$$

Conditional assignments are wrapped in an `if` statement:

```
$\mathbf{A}_2[assigns\,\mathbf{if}\,cond; lvalues] \rightarrow \mathbf{if}\ (\mathbf{ALIAS}[cond])\ \{$
                                          $\mathbf{A}_1[assigns; lvalues]$
                                      $\}$
```

Quantified assignments involve the use of control variables over which we quantify expressions. We translate these by first declaring the control variables, then recursively turning each quantifier (which is of the form $e_1\ op_1\ v\ op_2\ e_2$) into a for-loop. To compute the initial values and upper bound for the for-loop, we use $\mathbf{RANGE}_1$ and $\mathbf{RANGE}_2$, respectively, which are defined as:

$$\begin{aligned}&\mathbf{RANGE}_1[lv; e; <] \rightarrow lv = e + 1\\ &\mathbf{RANGE}_1[lv; e; \leq] \rightarrow lv = e\end{aligned}$$

$$\begin{aligned}&\mathbf{RANGE}_2[lv; e; <] \rightarrow lv < e\\ &\mathbf{RANGE}_2[lv; e; \leq] \rightarrow lv \leq e\end{aligned}$$

$$\mathbf{A}_3[cvars; quants; assigns] \rightarrow \mathbf{C}_1[cvars] @\ \mathbf{A}_{3,1}[quants; assigns]$$

```
$\mathbf{A}_{3,1}[(e_1\ op_1\ v\ op_2\ e_2, \mathbf{none}) :: quants; assigns] \rightarrow$
  $\mathbf{for}$ ( $\mathbf{RANGE}_1[\mathbf{ALIAS}[v]; \mathbf{ALIAS}[e_1]; op_1]$;
        $\mathbf{RANGE}_2[\mathbf{ALIAS}[v]; \mathbf{ALIAS}[e_2]; op_2]$;
        $\mathbf{ALIAS}[v] + +$) {
            $\mathbf{A}_{3,1}[quants; assigns]$
  }
```

$$\mathbf{A}_{3,1}[\mathbf{nil}; assigns] \rightarrow \mathbf{ASSIGN}[assigns]$$

If the quantifier includes extra conditions (such as in `<| i: 0<=i<10 & odd(i) :: a[i] = i >`), we wrap the final assignment clause in a conditional statement:

```
$\mathbf{A}_{3,1}[(e_1\ op_1\ v\ op_2\ e_2, cond) :: quants; assigns] \rightarrow$
  $\mathbf{for}$ ( $\mathbf{RANGE}_1[\mathbf{ALIAS}[v]; \mathbf{ALIAS}[e_1]; op_1]$;
        $\mathbf{RANGE}_2[\mathbf{ALIAS}[v]; \mathbf{ALIAS}[e_2]; op_2]$;
        $\mathbf{ALIAS}[v] + +$) {
            $\mathbf{if}(cond)$ {
                $\mathbf{A}_{3,1}[quants; assigns]$
            }
  }
```

When the translation terminates, Phobos attempts to convert the final term to C abstract syntax, and the resulting OCaml structure is passed to Mojave for optimization and code generation.

4.4 Further considerations

Our use of the **CHANGED** variable to track changes in the program state is based on the assumption that executing all assignment statements must result in a state change unless the program has reached a fixed point. Although this only holds in the absence of randomness, we see the expected benefits in most programs. For instance, the sorting program shown in Figure 1 terminates after one iteration if the input array is initially sorted, while it takes N iterations for arrays sorted in reverse order.

Although we have described a concrete implementation for assignment translation, we can easily modify our approach to be more abstract. By overriding $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$ we may implement an alternative way of handling assignments. A simple modification would be to encode each assignment statement as a local function and store references to these functions in a global store. The main loop could then be modified to schedule assignment statements nondeterministically, by using a fair random number generator to determine the statement ordering and tracking state changes in a more sophisticated way.

We could also replace the main program loop with a call to a generic scheduler. This could be implemented as a C function to be linked against when compiling UNITY programs, providing full customization of the scheduler.

In addition, we can easily translate to any target abstract syntax supported by the Mojave compiler by modifying the various $\mathbf{A}$ and $\mathbf{C}$ operators in our implementation. Source-to-source translation from any target abstract syntax is available using Mojave's pretty-printing capabilities.

5 Conclusion

This paper has presented a method of translating UNITY programs into executable code, using term rewriting as an integral part of the compilation process. Our method has several advantages over other techniques for compiling UNITY programs, including easy translation to multiple languages and the ability to change the scheduler for UNITY statements. On the other hand, we have ignored rewriting termination or Church-Rosser properties of our implementation.

We intend to exploit Mojave's integrated MetaPRL theorem prover to carry out the derivation of formal proofs for properties of UNITY programs, and to apply our translation method to additional UNITY-based formalisms. Examples of such formalisms are the Communications and Control Language (CCL) [13], which we are using to specify and implement programs for multi-vehicle control systems [4], and Dynamic UNITY [18], a specification language and logic for message-passing systems that exhibit dynamic behavior (such as process creation and deletion).

We believe that translations from these formalisms can be carried out using methods similar to those we have applied to UNITY. The translation of Dynamic UNITY, in particular, will require significantly more runtime machinery than we have provided for UNITY programs; the presence of a weakly fair scheduler suffices to execute a UNITY program, but the execution of a Dynamic UNITY system requires additional constructs such as process tables and message queues.

References

1. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990.
2. F. Andersen, K. Petersen, and J. Petterson. Program verification using HOL-UNITY. In *International Workshop on Higher Order Logic and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 1–16. Springer–Verlag, Heidelberg, Germany, 1994.
3. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison–Wesley Publishing Company, Reading, MA, USA, 1988.
4. Lars Cremean, William Dunbar, David van Gogh, Jason Hickey, Eric Klavins, Jason Meltzer, and Richard M. Murray. The Caltech multi-vehicle wireless testbed. In *Proceedings of the Conference on Decision and Control (CDC)*. IEEE Press, 2002. `http://www.cs.caltech.edu/~mvwt/`.
5. D. C. DeRoure. Parallel implementation of UNITY. In *The PUMA and GENESIS Projects*, pages 67–75, 1991.
6. M. J. C. Gordon and T. F. Melham. *Introduction to HOL—A theorem proving environment for higher order logic*. Cambridge University Press, Cambridge, England, 1993.
7. A. Granicz and J. Hickey. Phobos: A front-end approach to extensible compilers. In *36th Hawaii International Conference on System Sciences*. IEEE, 2002.
8. B. Heyd and P. Cregut. A modular coding of UNITY in Coq. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 1996.
9. Jason Hickey and Alexey Nogin. Fast tactic-based theorem proving. In *Theorem Proving in Higher Order Logics (TPHOLs 2000)*, August 2000.
10. Jason J. Hickey. Nuprl-Light: An implementation framework for higher–order logics. In *14th International Conference on Automated Deduction*. Springer, 1997.
11. Martin Huber. MasPar UNITY version 1.0. `ftp://sanfrancisco.ira.uka.de/pub/maspar/maspar_unity.tar.Z`, 1992.
12. J. Hickey, J. D. Smith, A. Granicz, N. Gray, C. Tapus, and B. Aydemir. The Mojave Research Group Website. `http://mojave.cs.caltech.edu/`.
13. Eric Klavins. Communication complexity of multi-robot systems. In *Fifth International Workshop on the Algorithmic Foundations of Robotics*, Nice, France, December 2002. Proceedings to appear as a book in the Springer-Verlag series on advanced robotics.
14. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In *Theorem Proving in Higher-Order Logics (TPHOLs '02)*, 2002.
15. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer–Verlag, Heidelberg, Germany, 1994.
16. Lawrence C. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32, 2000.
17. S. Radha and C. R. Muthukrishnan. A portable implementation of UNITY on Von Neumann machines. *Computer Languages*, 18(1):17–30, 1993.
18. Daniel M. Zimmerman. *Dynamic UNITY*. PhD thesis, Department of Computer Science, California Institute of Technology, 2001.