Proposal

ARPA/ITO BAA 98-10
FRAMEWORK FOR INTEROPERABLE REPRESENTATIONS

A Reflective Framework for Formal Interoperability

Submitted by:

SRI
Computer Systems Laboratory
333 Ravenswood Ave
Menlo Park CA 94025

In Collaboration with:

Department of Computer Science, Stanford University
Stanford, California 94305

A. Innovative Claims

In practice we find ourselves in constant need of *moving back and forth* between different formalizations capturing different aspects of a system. For example, in a large software system we typically have very different requirements, such as functional correctness, performance, real-time behavior, concurrency, security, and fault tolerance, which correspond to different views of the system and that are typically expressed in different formal systems. Often these requirements affect each other, but it can be extremely difficult to reason about their mutual interaction, and no tools exist to support such reasoning. This situation is very unsatisfactory, and presents one of the biggest obstacles to the use of formal methods in software engineering because, given the complexity of large software systems, it is a fact of life that no single perspective, no single formalization or level of abstraction suffices to represent a system and reason about its behavior. We need (meta-)formal methods and tools to achieve *Formal Interoperability*, that is, the capacity to move in a mathematically rigorous way across the different formalizations of a system, and to use in a rigorously integrated way the different tools supporting these formalizations.

We will develop new, formal interoperability methodologies and *generic* meta-tools that are expected to achieve dramatic advances in software technology and formal methods:

- A general methodology for defining inter-language translations, where the languages can be architectural description languages (ADLs), formal specification languages, or programming languages, and where the mappings may move across these different categories of languages.
- A formal language translation meta-tool that, given formal specifications for two languages and their mapping will generate an executable translator tool transforming statements and modules in one language into their counterparts in the other language.
- A do-it-yourself methodology for generating semi-automatically a reasonably efficient theorem prover implementation from a meta-logical specification of the theorem prover's inference system; and for easily combining different theorem provers defined with this method.
- A general methodology for creating and checking *heterogeneous proof objects* that can certify the correctness of the final result of a proof effort involving different theorem provers working in a distributed way.
- A heterogeneous meta-proof checker tool, that, given meta-logical specifications of several formal systems and their inter-translations will generate a proof checker for heterogeneous proofs in the combined system.
- A general methodology for endowing formal specification languages and declarative programming languages with powerful and extensible module composition mechanisms in a formalism-independent way.
- A module calculus meta-tool that, given a logic-based specification or programming language satisfying reasonable requirements will generate an executable language extension of it, enjoying powerful and extensible module composition operations.
- A metaprogramming methodology using modules reified as data, for reflective adaptation of software modules at runtime that may depend on changes in their environment.

B. Technical Rationale

B.1 Introduction

In practice we find ourselves in constant need of *moving back and forth* between different formalizations capturing different aspects of a system. For example, in a large software system we typically have very different requirements, such as functional correctness, performance, real-time behavior, concurrency, security, and fault tolerance, which correspond to different views of the system and that are typically expressed in different formal systems. Often these requirements affect each other, but it can be extremely difficult to reason about their mutual interaction, and no tools exist to support such reasoning. Furthermore, to manage the complexity and evolution of complex systems it is important to have formal representations at different levels of abstraction such as architectural descriptions, formal specifications, finite state models, and code structure. Again it is important to be able to move rigorously across levels and to understand precisely what constraints are imposed by descriptions at the different levels upon one another.

At present, formal methods for software specification and verification are *monolithic*, in the sense that in each approach only one formal system or specification language is used to formalize the desired system properties. For this reason, although formal approaches are very useful at their appropriate level of abstraction, their formal systems, and the tools based on them, are as it were *autistic*, because they lack the theoretical basis for being related to other formalisms and to their supporting tools. As a consequence, it is at present extremely difficult to integrate in a rigorous way different formal descriptions, and to reason across such descriptions. This situation is very unsatisfactory, and presents one of the biggest obstacles to the use of formal methods in software engineering because, given the complexity of large software systems, it is a fact of life that no single perspective, no single formalization or level of abstraction suffices to represent a system and reason about its behavior.

We need (meta-)formal methods and tools to achieve *Formal Interoperability*, that is, the capacity to move in a mathematically rigorous way across the different formalizations of a system, and to use in a rigorously integrated way the different tools supporting these formalizations. Meta-tools of this kind may include:

- **meta-logical frameworks**, in which many different logics and specification languages can be defined, executed, and interoperated.
- **meta-logical translators**, in which rigorous translations between any given pair of logical formalisms can be defined and automated. This can be extremely useful to vastly increase the applicability of tools, for example, to use a theorem prover for one logic to prove theorems in a different logic.
- **meta-logical module calculus tools**, which given specifications in an arbitrary logic, or in several logics, can support powerful parameterization and module composition operations to combine those specifications in a highly modular and reusable way.

Crucial to the attainment of formal interoperability is the issue of correctness. To understand what correct formal interoperation means, key meta-level notions such as logical formalism, mappings between formalisms, reflective logical frameworks, and heterogeneous composition must be made mathematically precise.

The proposed meta-tools and technology for formal interoperation will therefore be based on a solid formal foundation and methodology. This will provide a flexible and extensible framework to support building, converting, composing, and analyzing multiple models of software and systems operating at different levels and having different underlying formalisms.

B.2 Core Technologies

In the following we briefly sketch recent work on meta-logical foundations that provide the basis on which to achieve the goal of formal interoperability. Specificaly, we briefly discuss:

- the theory of *general logics*, and the associated notion of *mapping* between logics,
- the notion of a *meta-logical framework*, and the use of *rewriting logic* for this purpose,
- the Maude implementation of rewriting logic,
- the notion of *reflection* and the power of reflective techniques, and
- *Open Mechanized Reasoning Systems* as a framework for formal interoperation of reasoning tools.

General Logics and Mappings

No single logical formalism (specification language, prototyping language, etc.) will be best for all purposes. What exists is a space of possibilities in which careful choice of the formalisms that best suit some given purposes can be exercised. The theory of *general logics* first proposed in [117] is aimed at analyzing this space of logics and providing a basis for general logic-independent tools. The theory is modular in that it permits focusing on different aspects of a logic—such as the provability relation, the models, or the proofs themselves—as well as combining several aspects depending on the particular applications. The theory provides axiomatic characterizations of the different aspects of a logic and of their combinations into fuller notions of logic. These different aspects are axiomatized by means of the following basic concepts from [117]:

- *an entailment system* axiomatizes the consequence relation of a logic
- *an institution* (a notion due to Goguen and Burstall [57]) covers the model-theoretic aspects of a logic, focusing on the notion of validity
- *a logic* is obtained by combining an entailment system and an institution
- *a proof calculus* enriches an entailment system with an actual proof theory
- *a logical system* is a logic with a choice of a proof theory for it

What is crucial in practice is to have correct and flexible methods for changing our point of view, so that we can move from one model or formalism to another. This is what maps of logics make possible. A *mapping* of logics translates one logic into another

$$\mathcal{L} \longrightarrow \mathcal{L}'$$

preserving whatever logical properties are relevant, such as provability of formulas, or satisfaction of a formula by a model. Such mappings allow us to relate different logics in a rigorous way, to combine different formalisms, and to explore new logics for computational purposes. This can serve as a foundation and methodology to achieve *formal interoperability*, that is, the capacity to move in a mathematically rigorous way across the different formalizations of a system, and to use in a rigorously integrated way the different tools supporting such formalizations.

The definition and automation of maps of this kind will be supported by tools such as a *meta-logical translator* that, when suppied with the formal definition of a given map of logics, will implement such a map. Such a meta-tool will be very useful to in enabling the formal interoperation of tools in different formal systems. For example, it could be used to automatically translate specifications in one logic into corresponding specifications in another for which a theorem prover exists. Then, if the map has the required properties—namely, if it is *conservative* [117]—such a theorem prover could be used to mechanically prove theorems about the source specification.

Meta-Logical Frameworks and Rewriting Logic

A *meta-logical framework* $\mathcal{F}$ is a "universal logic" into which a wide variety of other logics can be faithfully represented by means of conservative maps of logics

$$\mathcal{L} \longrightarrow \mathcal{F}.$$

A number of logics have been proposed in the recent literature as (meta-)logical frameworks of this kind. Several of these proposals offer some variant of higher-order type theory as the framework logic (see [97] for references). A simpler alternative recently proposed by Martí-Oliet and Meseguer [96, 97] is to use *rewriting logic* [119]. Rewriting logic is a very flexible, yet simple, logic that can be naturally used as a general logical and semantic framework, that is as a *metalogic* [96] in which:

- other logics can be defined;
- languages can be semantically defined and can be prototyped;
- heterogeneous languages and systems can be integrated; and
- generic tools can be designed with a precise formal basis.

In rewriting logic, axioms are rewrite rules of the form

$$t \rightarrow t'$$

that can be applied *modulo* given structural axioms specified with the syntax. Such rewrite rules have both a logical and a computational reading. Logically, each rewriting step is a logical *entailment* in a formal system. Computationally, each rewriting step is a *parallel local transition* in a concurrent system. Therefore, rewriting logic can be used both as a *meta-logical framework* in which to represent other logics, and as a *semantic framework* for defining and prototyping computational systems and languages. And by using an implementation of rewriting logic such as the Maude system these logics can be prototyped and executed by means of their corresponding rewriting logic representations. Many logics—for example, first-order logic, intuitionistic logic, linear logic, equational logic, Horn logic with equality, and several sequent-based logics—have been represented in a very simple and natural way in rewriting logic. Indeed, we conjecture that any effectively presented logic—that is, any logic of practical computational interest—could be represented and prototyped in rewriting logic in this way. Furthermore, using reflective techniques it is possible to define and execute in rewriting logic not only representations of one logic $\mathcal{L}$ into the framework of rewriting logic, but also general mappings $\mathcal{L} \longrightarrow \mathcal{L}'$ from one logic into another.

Because of its great flexibility for representing computational systems, including concurrent ones, rewriting logic can also be used as a very flexible *semantic framework* in which formal models for many different languages and systems. For example, CCS, Petri nets, Actors, the UNITY language, functional languages, dataflow, graph rewriting, object-oriented languages, and conventional languages, can all be defined, prototyped and executed in rewriting logic.

Rewriting Logic and the Maude Language

A key advantage of rewriting logic as a logical and semantic framework is that rewriting logic specifications can be efficiently executed. This provides a general way to prototype and execute different logics, and also different models and languages—including both architectural description languages and programming language definitions. In all cases, we just execute their corresponding rewriting logic specifications, Furthermore, it is then easy to interoperate different logics or formal languages by combining their executable rewriting logic representations.

The Maude system [28, 121] is an efficient implementation of rewriting logic that combines the flexibility and traceability of an interpreter with the efficiency of advanced semicompilation techniques. It can reach 590,000 rewrites per second on relevant applications running on a Pentium PC. It supports sophisticated rewriting *modulo* a wide range of combinations of equational theories such as associativity, commutativity, identity, and idempotency; it has a very expressive type structure based on membership equational logic [123] that supports types, subtypes, subtype polymorphisms, and partial operations. Maude modules can import other modules and can be parameterized, thus also supporting an algebraic style of parametric polymorphism. A key additional feature of crucial importance for the applications envisioned in this proposal is that Maude is *reflective* and can support an arbitrary number of levels of reflection in its computations. This feature, further explained in what follows, is based on the fact that rewriting logic itself is reflective [30]. Furthermore, since rewriting logic is intrinsically concurrent, adequate subsets of the logic can be efficiently implemented as concurrent and mobile languages [89]. This provides a wide-spectrum formal framework to pass from executable specifications to correct and efficient code.

Reflection

Reflection is a system's ability to represent and control its own metalevel. In this way, very powerful techniques to extend, adapt, and modify a system become available. Many researchers have recognized the great importance and usefulness of reflection in programming languages [180, 178, 193, 190, 84, 75, 114], in theorem-proving [201, 18, 176, 79, 34, 112, 56, 73], in concurrent and distributed computation [110, 162, 171], and in many other areas such as compilation, programming environments, operating systems, fault tolerance, and databases (see [177, 83] for recent snapshots of research in reflection). In particular, reflective methods are enormously useful in achieving formal interoperability.

In spite of being a very important topic, the semantic foundations of reflection are not yet well understood. For the case of reflection in logical languages we have recently proposed a general axiomatic notion of reflective logic [29]. Our axioms are based on the theory of general logics [117]; they are formulated in a logic-independent way that makes the reflective logic an explicit parameter and does not depend on the particular details of each logic.

The key concept in our axiomatization of reflective logics is the notion of a *universal theory*, that is, a theory U that can simulate the deductions of all other theories in a class $\mathcal{C}$ of theories of interest. In particular, if U is one of the theories in the class $\mathcal{C}$, then U can simulate its own metalevel at the object level, and this process can be iterated *ad infinitum*, giving rise to a "reflective tower."

In [29] several examples of reflective logics, including the lambda calculus and Turing machines, are discussed. However, our original motivation for studying reflective logics has come from our work on rewriting logic and our desire of endowing the Maude language with reflective features in a rigorous way. In fact, we have proposed a specific universal theory for rewriting logic in [29] and have proved it to be correct [30], that is, that it satisfies all the formal requirements that we have stated for a reflective logic in [29].

Reflection is an enormously powerful and useful property in practice. Therefore, a key aspect of our research, along with the development of the theoretical foundations, has been exploiting reflection in a wide range of applications, using the Maude language [28], which implements rewriting logic and supports its reflective capabilities, as a tool for developing those applications.

A number of areas of application of reflection in rewriting logic and Maude are directly relevant for formal interoperability:

- *Logical framework applications*, to make executable the representations of many other logics within the logical framework of rewriting logic.
- *Modularity applications*, allowing the semantic definition and implementation of an open and extensible *module algebra* to combine program modules in very powerful ways.
- *Formal environments*. Using reflection, it becomes very easy to specify and build formal environments, providing theorem-proving and formal analysis support for a logically based specification or programming language. The idea is that the inference systems that these tools implement are themselves rewrite theories; the fact that such tools operate on metalevel entities—including the object theories that they verify or formally analyze—is naturally supported by the reflective features.

- *Adaptive software and metaprogramming.* Since modules are reified as data, it is possible and easy to write programs that manipulate, transform, and adapt other programs, including adaptation of software modules at runtime that may depend on changes in their environment.

Open Mechanized Reasoning Systems

Open Mechanized Reasoning Systems [53] is a general framework for modular construction and interoperation of mechanized reasoning systems. The ultimate goal of this work is to provide a framework and a methodology in which users, as well as system developers, can construct complex reasoning systems by composing existing modules, or add new modules to existing systems, in a "plug-and-play" manner. To realize this goal, reasoning modules are viewed as providers of *logical services* [182] (called OMRSs).

Three key components of an OMRS specification are: a *reasoning theory* component that is the counterpart of the logical notion of formal system, a *control* component that consists of a set of inference strategies, and an *interaction* component that provides an OMRS with the capability of interacting with other systems, including both other OMRSs and human users. Reasoning theories deal with the structure of assertions and rules for constructing *reasoning structures* (objects representing proof fragments). To address problems of scale and locality of information, reasoning theories can be structured as compositions of fragments (parameterized reasoning theories). Rules, called *bridge rules*, that relate assertions of different fragments are used to establish nontrivial links for interoperation of multiple theories. These bridge rules also serve as a formal basis for interfaces and specification of services *provided* or *required* by reasoning modules. To demonstrate the OMRS approach, a modular specification of the NQTHM simplifier as a collection of reasoning theories and bridge rules connecting the different reasoning experts is given in [33]. Based on this specification, a decomposition and reconstruction of the simplifier has been carried out and used to augment the prover with the ability to produce proofs (reasoning structures).

There is a strong connection between the OMRS framework and rewriting logic. Finitely presented reasoning theories correspond to a special class of rewrite theories, and reasoning structures generalize the proof terms of rewriting logic to term graphs which directly model sharing and partially developed proofs [115]. The control aspect of OMRS can be naturally represented as strategies in rewriting logic. Therefore, the OMRS framework can be naturally represented and executed in a rewriting logic language such as Maude.

Although the OMRS work is focused on interoperation and integration of mechanized reasoning systems, the problems addressed include general problems of modularization, composability, and interoperability that are not specific to the domain of mechanized reasoning. Thus, the ideas and techniques developed here can be applied to formal interoperability of other formalisms.

B.3 Technical Approach

This section describes our technical approach to the core facets of the proposed research. Our technical approach for the additional options is presented in § B.4. The proposed research will have three main interconnected facets:

- formal interoperability meta-tools, and
- formal interoperability case studies, leading to a
- formal interoperablity methodology.

These will be developed in parallel to gain the benefit of mutual feedback. The unifying idea is using reflection in rewriting logic for the meta representation of (representations) as Maude data types in which terms meta-representing theories can be executed or analyzed or can be transformed by simple user definable operations. We will initially focus our efforts on three tool categories:

- A meta-tool for mappings between formal languages, where the languages can be architectural description languages, formal specification languages, or programming languages, and where the mappings may move across these different categories of languages.
- Proof assistants – methods for defining reasoning tools in Maude and for interoperating Maude-based and external tools. The advantages of this approach include:
  - rapid prototyping of ideas,
  - interoperation of diverse tools, and
  - formally based tools that are easier to verify and to safely modify.
- A generic module calculus with sophisticated operations on terms that represent software modules.

Mappings

As explained above, rewriting logic is a logical framework in which many different logics, languages, and systems can be naturally represented. This is very useful for executable specification, rapid prototyping, and formal reasoning. Clavel and Meseguer have shown in [29, 30] that rewriting logic is reflective. That is, there is a rewrite theory $\mathcal{U}$ with a finite number of operations and rules that can simulate any other finitely presentable rewrite theory $\mathcal{R}$ in the following sense: given any two terms t, t' in $\mathcal{R}$, there are corresponding terms $\langle\overline{\mathcal{R}}, \overline{t}\rangle$ and $\langle\overline{\mathcal{R}}, \overline{t'}\rangle$ in $\mathcal{U}$ such that we have

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \longrightarrow \langle\overline{\mathcal{R}}, \overline{t'}\rangle.$$

To support efficient reflective computations, Maude builds in key features of the universal theory $\mathcal{U}$ in the form of a Maude module called `META-LEVEL`. In particular, in `META-LEVEL` there is a data type, `Module`, whose terms are meta-representations of Maude modules, that is, of theories in rewriting logic. In [99] a very general reflective method for reifying a representation map $\mathcal{L} \to \mathit{RWLogic}$ ($\mathcal{L}$ a logic, *RWLogic* rewriting logic) inside rewriting logic itself is proposed. This reification takes the form in Maude of an equationally defined function between the abstract data types representing the finitary theories of $\mathcal{L}$ and of the data type `Module` representing the finitely presentable theories of *RWLogic*.

One example of this powerful idea that has been worked out in detail is a complete specification of a reified map between full quantifier-free first-order linear logic and rewriting logic [99].

In this example, a linear theory T is defined as a finite set C of propositional constants together with a finite set S of sequents of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, where each A_i and B_j is a linear logic formula built from the constants in C. Note that with this definition, all linear theories are finitely presentable. First, we define an abstract data type `LL-ADT` to represent linear theories. A linear theory is represented as a term `< C | G >`, where `C` is a list of propositional constants (that is, identifiers), and `G` is a list of sequents written in the usual way. Having defined the abstract data types to represent both linear and rewrite theories, we define a function $\overline{\Phi}$: `LL-ADT` $\longrightarrow$ `Module` mapping a term $\overline{T}$ in `LL-ADT` representing a linear theory T to a term in `Module` representing the corresponding rewrite theory $\overline{\Phi}(\overline{T})$ that represents T. We can summarize the reification $\overline{\Phi}$: `LL-ADT` $\longrightarrow$ `Module` of the map of logics Φ : *LinLogic* $\longrightarrow$ *RWLogic* we have just defined by means of the following commutative diagram:

$$\begin{tikzcd} \texttt{LL-ADT} \arrow[r, "\overline{\Phi}"] \arrow[d] & \texttt{Module} \arrow[d] \\ \mathit{LinLogicTh} \arrow[r, "\Phi"] & \mathit{RWLogicTh} \end{tikzcd}$$

Since each term of the data type `Module` can be compiled and executed in Maude as a rewrite theory, the above mapping automatically provides an implementation for linear logic theories in Maude, by executing for each linear theory T its corresponding rewrite theory $\overline{\Phi}(\overline{T})$, where the term $\overline{T} \in$ `LL-ADT` represents T.

This method, which has already been implemented in Maude for the above example [31], is completely general, in that it can be applied to any effectively presented map of logics Ψ : $\mathcal{L} \longrightarrow$ *RWLogic* that maps finitely presentable theories in $\mathcal{L}$ to finitely presentable theories in rewriting logic. Indeed, the effectiveness of Ψ means exactly that the corresponding $\overline{\Psi}$: $\mathcal{L}$`-ADT` $\longrightarrow$ `RWL-ADT` is a computable function and therefore, by the metatheorem of Bergstra and Tucker [10], that it is specifiable by a finite set of Church-Rosser and terminating equations inside rewriting logic.

In fact, the meta-representation can be applied to translate not only logics and formal specifications, but also other formal languages such as architectural description languages and programming languages. In general, given two formal languages, $\mathcal{L}$ and $\mathcal{L}'$, we can meta-represent modules in these languages as data types $\mathcal{L}$`-ADT` and $\mathcal{L}'$`-ADT`. This provides an even more general method for defining and implementing any effectively presented map $\Phi : \mathcal{L} \longrightarrow \mathcal{L}'$ between two formal languages: one simply has to specify the function $\overline{\Phi}$: $\mathcal{L}$`-ADT` $\longrightarrow$ $\mathcal{L}'$`-ADT` by a finite set of Church-Rosser and terminating equations that always exists if $\overline{\Phi}$ is effective. This therefore provides a very general method for implementing maps of formal languages and can be used as the technical core of a meta-logical translation tool. Such a tool will allow a user to:

1) specify the syntax and modules of two different formal languages, $\mathcal{L}$ and $\mathcal{L}'$,

2) define a map $\Phi : \mathcal{L} \longrightarrow \mathcal{L}'$,

3) execute such a map, to translate $\mathcal{L}$-modules into $\mathcal{L}'$-modules,

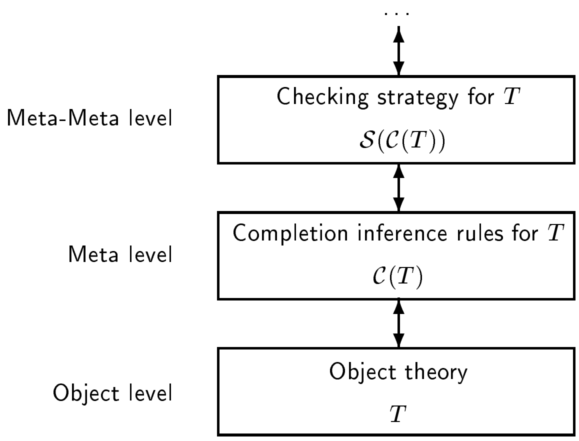
4) in case $\mathcal{L}'$ is *RWLogic*, to execute translated modules representing those in $\mathcal{L}$.

Proof Assistants

Reflection can also be exploited to make the development and interoperation of new formal tools for any logic of interest much easier. The idea is that, with the help of reflection, such tools can themselves be easily specified and implemented in rewriting logic with remarkable efficiency.

We have recently demonstrated the many advantages of this reflective method of building proof assistants—including the ease of programming, understandability, maintainability, and interoperability of the tools so developed, with at least one order of magnitude gain in software productivity compared with tool building in a conventional language—by means of two tools that will be freely available on the web as part of the Cafe project in Japan, namely a Knuth-Bendix checker tool and an inductive theorem prover for order-sorted equational logic [22]. We briefly explain the reflective design of these tools here.

The goal of the Knuth-Bendix checker is to ensure that a given equational theory is *decidable* because its equations, when used as rewrite rules, always terminate with a unique result or canonical form. After checking termination, the uniqueness of the canonical form (called convergence) can be checked by checking that the equational theory T is already complete under certain inferences expressed by a completion inference system, $\mathcal{C}$. The theory T that we want to complete appears at the object level. The inference rules $\mathcal{C}$ are clearly metalevel rules defining a theory one level up. The key point is that the strategy controlling the application of such inference rules, and therefore yielding our desired checking procedure, appears as a term in the theory $\mathcal{S}(\mathcal{C}(T))$, corresponding to instantiating an internal strategy language $\mathcal{S}$ with the inference rules $\mathcal{C}$ as its object theory. Since $\mathcal{S}$ transforms any theory into a metalevel theory, and $\mathcal{C}$ is already a metalevel theory, $\mathcal{S}(\mathcal{C}(T))$ is a theory in the meta-metalevel. Keeping these three levels of reflection separate is the key to maintaining modularity. This is illustrated in the figure below.



The reflective design of the inductive theorem prover tool is quite similar to that of the Knuth-Bendix checker tool. There are again three levels. At the object level we have the theory T about which we want to prove some theorems. The inductive inference rules use T as data and are therefore at the meta-level. Finally, the theorem-proving *strategies* that one may want to use to apply the inductive inference rules in a specific way to prove a theorem are at the meta-meta level.

As part of the proposed research we plan to further develop and illustrate this reflective methodology for building proof assistants. Specifically, we will develop several new tools, including a Knuth-Bendix completion tool,[1] and a coherence completion tool [192] to complete rules in a rewrite theory relative to equations in its equational subtheory, also assuming equations in membership equational logic. We will also extend and generalize the current inductive theorem prover tool so as to support inductive proofs in both membership equational logic and rewriting logic as well as proofs of sufficient completeness for specifications. Furthermore, we will specify a temporal logic and select both a model-checker and a general proof system for that logic (possibly Murphi and STEP, respectively), to demonstrate the combined power and interoperation of the new tools in conjunction with an already existing model-checking tool. Ease of interoperation and ease of tool development are two of the key features that will be demonstrated in this combination of tools.

The specification and implementation of these proof tools will be based on a general do-it-yourself methodology for semi-automatically generating reasonably efficient implementation of proof systems from a meta-logical specification of the prover's inference system, and for easily combining different theorem provers defined with this method. Translations between logics and theories will provide the basis for semantically meaningful combination of proof systems. These experiments will demonstrate both the methodology of building efficient reasoning components and the ease of combining such components with each other and with preexisting reasoners, using the Maude tool.

An important advantage resulting from having proofs as abstract objects is that proofs can be constructed and passed along with the subject of the proof (code or theorem, for example) to potential users. The user can be assured of the required property not merely by a claim from the provider, but also by checking the proof, which is generally quite simple. Using the meta-logical framework of rewriting logic there is a general methodology for creating and checking *heterogeneous proof objects* that can certify the correctness of the final result of a proof effort involving different prover systems working in a distributed way. This methodology is based on the algebra of proof terms derived from the rewrite rule specification of each proof system's inference rules. The OMRS notion of bridge rules to combine both reasoning theories and reasoning structures can be used to specify the relations between assertions in different proof systems and thus form the basis for a notion of heterogeneous proof terms that combines proofs from different proof systems to obtain overall proofs. The bridge rules in turn are based on translations between the underlying formal systems.

This methodology can also be used to produce high-level proofs resulting from the combination of existing provers. The prover can be considered simply as a black box implementing the side condition on the rule that corresponds to the prover. A deeper analysis of the provers inference engine would provide the ability to enhance the prover's ability to produce more detailed justifications.

Interoperation of Architecture Description Languages

The idea of interoperating diverse logical formalisms and tools by combining their executable representations in rewriting logic also applies to other classes of languages. We have shown by

[1] The tool already developed for Cafe provides only the more limited capability of *checking* whether a given specification is Church-Rosser, and does so for the more restricted case of order-sorted equational specifications. The new tool will provide full Knuth-Bendix completion capabilities for the more general and expressive case of membership equational logic[123, 11].

example how components expressed in different architectural styles can be combined and executed in Maude [116]. This was just an instance of a more general approach in which ADLs can be represented as data-types, mappings defined from components described in a given ADL to rewrite theories in an appropriate semantic model, and connectors mapped to adaptor modules. Different ADLs may use different formalisms to specify component and connector interfaces, such as CSP, Lotos, automata formalisms, and temporal logics. By using general mappings from Maude representations of these formalisms to tools for analyzing and checking properties, tools for different formalisms can be applied to heterogeneous architecture descriptions; and by using formal mappings between formalisms their results can be combined in semantically meaningful ways.

Meta-logical Module Calculus

Perhaps one of the most important new contributions of Maude is the *metaprogramming methodology* that it supports in a simple and powerful way. This methodology is well integrated with the language's semantic foundations, particularly with its logical foundations for reflection.

By "metaprogramming" we of course mean the capacity of defining programs that operate on other programs as their data—in our case, equational and rewrite theories that operate on other such theories as their data. By observing that we cannot only reify theories, but also views among them, this includes the more traditional "parameterized programming" capabilities in the Clear-OBJ tradition [20, 66] as a particular instance. The difference is that traditionally theories are metalevel entities not accessible at the object level of the logic, since this is only possible in an explicitly reflective logical context.

What reflection accomplishes is to *open up to the user* the meta-level of the language, so that instead of having a fixed repertoire of parameterized programming operations we can now define a much wider range of theory-transforming and theory-combining operations that could not be defined through more traditional means. In summary, an important capability that reflection makes possible in Maude is the definition of an open, extensible, and user-definable *module algebra* supporting a new style of metaprogramming with very promising advantages for software methodology.

This capability is important because large software systems are in constant need of *evolution*. They need to be extended and modified in ways that were not anticipated when a system was originally built, because they exist in a social context with evolving needs that generate new system requirements. Present software technology provides quite limited support for reusing, adapting, and modifying large systems. Although important progress has been made in the area of ADLs, such languages remain for the moment primarily syntactic in nature. The mathematical semantics of software composition is important in specifying the conditions under which composition operations will behave properly; it is also important as a formal framework in which different ADLs and different module composition mechanisms can be defined and related in a rigorous way.

To develop new software technology capable of drastic improvements in this area, a promising research direction is developing a *module calculus* for composition and compact formal description of software systems and software designs with the following characteristics:

- *Wide-Spectrum.* With module composition operations simultaneously transforming specifications, prototypes, and code.

- *Semantics-Based.* Formally stating the semantic requirements that must be met in order for the corresponding compositions to be correct.
- *Language-Independent.* So that the calculus can be applied to many languages.
- *Formalism-Independent.* So that it can be applied to different specification formalisms.
- *Extensible.* So that new module composition operations can easily be added to the calculus to support the widest possible variety of system adaptations and reuses.

Developing a module calculus with these features is a challenging research problem requiring new foundations. Fortunately, a body of existing work can be used as a basis. At the level of specifications, formalism-independent techniques going back to the institution ideas of Goguen and Burstall [57] and further extended in the theory of general logics [117], discussed above, have already been proposed, and very encouraging experience already exists in this regard in specification languages such as Clear [20], OBJ [66], ACT [39], Eqlog [61], and Maude [121]. These techniques have also influenced the module mechanisms of ML, and have syntactic-based counterparts in languages such as C++ and Ada.

The above-mentioned body of work is not sufficient to make the module calculus truly extensible. A reflective formal framework such as rewriting logic is essential for this purpose.

A particularly important pragmatic aspect of a modular specification concerns whether or not it is stable under evolution and extension: when the specified system is to be changed or extended, the changes or extensions to the specification should preferably be commensurate. This feature is concerned more with the contents of the modules (their interfaces) than with the module algebra used to compose them. Its presence ensures that the specification scales up smoothly as the size of the specified system increases.

Such good pragmatics have been achieved for the semantic specification of programming languages with the action semantics approach, developed by Mosses and Watt [158, 147, 198]. For example, when extending a functional (or imperative) programming language with concurrency constructs, the specification of the original language remains stable and merely has to be extended by the specification of the new constructs [147, 157]. This is in marked contrast to conventional denotational semantics, where such an extension would require reformulation of the entire specification.

The disciplined use of monads and monad transformers, as proposed by Moggi and others [133, 134, 40, 88, 87], also provides a systematic approach to achieving good pragmatics of modularity in denotational semantics; it is interesting that monads also support reflection [47]. Recently, a close relationship between action semantics and the monadic style of denotational semantics has been shown [195, 196].

B.4 Technical Approach for Options

Option 1. A Reasoning Tool for Java

For formal interoperation of tools that manipulate, analyze and transform code it is important to have a clear semantic basis. Modern programming languages such as Scheme, ML, C++, and Java provide both imperative and higher-order functional programming facilities. There is substantial work on developing a foundation for semantics-based manipulation of such languages.

Uniform semantics [183] is a unified framework for reasoning about program equivalence in lambda-languages — higher-order programs with effects. This includes imperative functional languages such as Scheme, Lisp, and ML as well as object-oriented languages such as Java. It places the methods developed in [184, 101, 102, 185, 104, 105, 77, 106], on semantics of imperative functional programs, in a more abstract and general setting. A lambda-language is a programming language obtained by augmenting the lambda calculus with primitive operations. The primitive operations that we have in mind include not only basic constants, branching, and algebraic operations such as arithmetic and pairing, but also operations that manipulate the computation state (store, continuation) and the environment (sending messages, creating processes). Uniform semantics is given by a syntactic reduction system satisfying certain uniformity constraints. There are general principles for reasoning about program equivalence in any sequential lambda language with uniform semantics combining the benefits of reduction calculi (modular axiomatization), and operational equivalence (more equations). Future work includes the extension of these results to concurrent languages, generalizing the theory of program equivalence for lambda-languages with primitives for creation of concurrent objects, and asynchronous message passing developed in [4].

Action semantics [158, 147, 151, 154] is a framework for modular description of programming languages, proposed by Mosses (in collaboration with D. A. Watt) about 10 years ago. Its main advantage over other frameworks is pragmatic: action-semantic descriptions have inherently good modularity, and scale up smoothly from smaller examples to realistic programming languages [197, 159, 67, 68, 161, 157]. Action Semantics uses semantic equations to give inductive definitions of compositional semantic functions that map abstract syntax to semantic entities. The semantic entities are primarily so-called *actions*, rather than the higher-order functions on Scott domains conventionally used in denotational semantics. Actions are expressed in a standard *action notation* that provides primitives and combinators for expressing flow of information and control. Action notation has itself been given an operational semantics [147, App. C]; this induces an operational semantics for any language from its ASD. The semantic entities also involve data, which is algebraically specified, and essentially a parameter of action notation.

Action semantics has recently been shown to be closely related to monad transformers [195, 196]. Moreover, although action notation already provides some support for reflection, the moadic treatment of reflection by Filinski [47] may lead to an enhancement of this aspect.

The frameworks of action semantics, uniform semantics, and rewriting logic have considerable potential for synergy. Adopting uniform semantics for defining action notation would make the theory of equivalence developed for uniform semantics applicable to action notation. By specifying uniform semantics in rewriting logic, existing tools for rewriting logic could be used for interpreting uniform semantics. Reduction of programming languages to action notation (as provided by existing and future action-semantic descriptions) would then lift uniform semantics to these languages. Finally, composing the mappings from action semantics to uniform semantics and from uniform semantics to rewriting logic should provide the basis for a useful semantics-based prototyping system. The Maude tool is an ideal medium for this activity. The advantages of this approach to formal tools for programming languages include:

- executable semantics;
- rapid prototyping and testing of ideas;

- extensibility—one can move gracefully from sequential to concurrent or reflective languages; and
- formal support for manipulating and reasoning about software and interoperation of tools.

To demonstrate the approach in a convincing way we propose to apply our methodology to Java, beginning with a clean but useful sequential fragment and later extending to account for concurrency and mobility. The development of such a Java tool will build on work defining the semantics of a simple object-oriented sublanguage of Java that has been written by Watt [199]. The existing prototyping tools for action semantics do not support concurrency, and the proposed tools based on Maude are needed for the validation of our proposed extension of Watt's semantics.

The project will benefit substantially from the participation of Professor Mosses who will be visiting our research group. The work will also benefit from the ongoing collaboration with Professor Ian Mason. A proposal for additional funding to continue the collaboration with Mason on this work will be submitted to the Australian Research Council.

Option 2. A Reflective Model for Adaptive Systems

One goal in this optional task is to develop the core technology of a new, formally based approach to runtime adaptive software using the reflective and meta-programming capabilities of rewriting logic. These capabilities are efficiently supported by the Maude system [28, 121], which will provide the experimental infrastructure of our research.

Our technical approach can be summarized as follows. We can think of an adaptive software component as a parameterized family of components, $\{C(p_1, \ldots, p_n)\}_{(p_1,\ldots,p_n)\in P}$ where the $(p_1, \ldots, p_n)$ are parameters that, when fixed, provide different adaptations of the component to different situations within a general parameter space of possibilities P. Corresponding to such an adaptive component there is a *formal model* taking the form of a parameterized family of executable specifications $\{M(\overline{p}_1, \ldots, \overline{p}_n)\}_{(\overline{p}_1,\ldots,\overline{p}_n)\in\overline{P}}$ in rewriting logic, together with an *explicit representation* of the relationship between the adaptive component and its formal model, which can be captured as a mapping $\phi : \overline{P} \longrightarrow P$ between the parameter spaces of the model and of the adaptive components, so that the model $M(\overline{p}_1, \ldots, \overline{p}_n)$ corresponds to the component $C(\phi(\overline{p}_1, \ldots, \overline{p}_n))$.

To complete a formal foundation for adaptive software, we need to develop an axiomatic characterization of computational reflection and its relation to logical reflection. We can then use the runtime behavior of $C(\phi(\overline{p}_1, \ldots, \overline{p}_n))$ and the reflective capabilities of rewriting logic to adapt the component with a new model $M(\overline{p}'_1, \ldots, \overline{p}'_n)$ and a new instantiation $C(\phi(\overline{p}'_1, \ldots, \overline{p}'_n))$ on the fly. This process of runtime adaptation can itself be specified at the meta-level as a meta-programming rewrite theory in Maude.

A natural next step in the development of this core technology is moving from adaptation of a single component to *distributed adaptation* of a collection of interacting components. We anticipate that the very good support provided by rewriting logic and Maude for distributed object-oriented reflection, including the important case of actor reflection, will prove key in this regard. Important kinds of distributed adaptation include adaptation to security threats and to failures, that can be dealt with in a modular and flexible way at runtime by means of the object-oriented reflective techniques that we plan to develop. We will use Maude to develop examples and case studies of adaptation and distributed adaptation. The ultimate fruit of this experimental work will be a formally based methodology for software adaptation of general applicability to a wide range of situations.

Option 3. Interoperable Formal Tools for OOD

Diagrams are often used in many aspects of the design of computer systems. Typical examples include data flow diagrams, flow charts, structure charts, entity-relationship diagrams, state diagrams, and Petri nets. Many popular object-oriented methods use different types of diagram to represent a system from multiple viewpoints. These methods can be particularly helpful in the early phases of requirements analysis and design by providing notations that are suitable for organizing the large volume of information that is used in system development. These modeling techniques greatly assist in obtaining an initial structuring of the design information. However, detailed problem analysis and design requires a degree of rigor not typically available from these informal methods.

There is great potential benefit in providing object-oriented design notations with formal semantics. Furthermore, if *executable* formal specifications can be systematically associated to graphical object-oriented descriptions it becomes possible not only to formally analyze and reason about such designs, but also to symbolically simulate their behavior, something not possible with the graphical descriptions alone.

OMT comprises three complementary models, object, dynamic, and functional models that are defined in terms of entity-relation diagrams, state diagrams, and data flow diagrams, respectively. These three diagrams respectively model the architecture (static), behavior (dynamic), and data flow and services of a given system. Cheng has developed formalization rules for two of the three OMT models, namely, the object model defined in terms of algebraic specifications [13, 26] and the dynamic model defined in terms of process algebras [194]. She has also developed a number of graphical-based tools to support specification construction and manipulation [12, 86, 25]. Wirsing [202] extends Jacobson's object and interaction diagrams with formal annotations. From the annotated diagrams, formal specifications, written in Maude, can automatically be generated and Java code can be generated from the Maude specification. Our approach is to provide formal semantics for object-oriented analysis and design (OOAD) notations, such as OMT (Object Modeling Technique), or the Jacobson notations, using rewriting logic as the semantic framework. The expected benefits include:

- semantic integration of the different viewpoints used in OOAD, both within one methodology and across methodologies, captured in terms of a single formal specification language
- a path from graphical descriptions to executable formal specifications (using the Maude tool for rewriting logic) that would enable rapid prototyping and provide early feedback to the developer
- semantics-based analyses, beginning with consistency and completeness
- a starting point for the application of formal refinement and code generation techniques.

Option 4. Designs for Logic Mappings

The goal of this task is to demonstrate the usefulness of our formal interoperability methodology and tools by showing how they can enhance exisiting theorem proving systems. Specifically, by providing interoperability-enabling mappings in or out of such systems, new applications and capabilities become possible. We will focus on the design of a mapping out of PVS and of a mapping into NuPrl as useful and important case studies. The research will be focused on the design of those mappings according to our proposed methodology; this will then serve as a basis for their future full implementations.

Specifically, we will focus on:

- A preliminary design of a translation from a subset of the PVS language to Maude. The expected benefit from a future implementation of this translation is a dramatic speeding up of the simplification and rewriting capabilities of PVS. This will be very useful in many application areas such as hardware verification, where one of the current key limitations is the computation time required for large applications. The high performance of the Maude rewrite engine (590,000 rewrites per second on a Pentium PC) combined with the traceability of those rewrites—so that proof objects can still be extracted if desired—seems to offer an excellent opportunity for exploiting formal interoperability in this concrete situation.

- A preliminary design according to our general methodology of a translation from HOL to NuPrl. The benefit of this is making available to NuPrl a large library of HOL theories and specifications. Although a translation from HOL to NuPrl has already been proposed by Howe [78]—who stresses the advantages of making in this way avalilable to NuPrl a large body of HOL specifications—it is not clear at present how such a translation can be seen as a particular instance of our proposed general methodology for translations between formal systems. This therefore will be an interesting test case for our methodology.

Option 5. Optimizing Adaptive Reflective Systems

Once a general reflective model of adaptive software and a corresponding methodology have been developed in Years 1 and 2, the next natural step will be to investigate (in Year 3) software optimization techniques making possible highly efficient, yet powerful, adaptive computation in practice.

The point is that reflection in general and adaptation in particular are powerful capabilities that come at a cost, including a cost in execution when they are performed at runtime. However, this cost can be drastically reduced – making the techniques competetive in practice – by means of a number of optimization techniques, including the following:

- compiling away levels of reflection by techniques such as supercompilation [190] and partial evaluation [108, 109]

- compiling away execution strategies and scheduling and runtime policies [164]

- compilation and evaluation of module expressions corresponding to complex instantiations of different combinations of adaptive components

- extension of all these techniques to a distributed object-oriented setting.

A study of these optimization techniques, their application to improving the efficiency of adaptive components, experimental evaluation of the efficiency thus gained through several case studies, and the emergence of new optimization techniques and methods for adaptive software are the results expected at the end of this one-year task.

G. Comparison with Ongoing Research

The core technology of general logics [117] extends the theory of institutions [57] and is related to other institution-like notions proposed in the literature, including [5, 172, 38, 173, 113, 168, 45, 44, 70] (see [126] for a discussion of work in this quite active field). Besides being generally more flexible than these other alternative notions in representing and integrating the different model-theoretic and proof theoretic components of a logic, the main advantage over these other approaches is its use in conjunction with a *reflective* logical framework such as rewriting logic. The essential difference is that, using reflection, metalevel notions such as mappings between logics that can only be theoretically defined in the other approaches can now be defined in an *executable* way within rewriting logic.

Our proposal of using rewriting logic as a logical framework is related to other logical framework proposals such as the Edinburgh logical framework LF [69, 6, 51]; meta-theorem-provers such as Isabelle [165], λProlog [160, 43], and Elf [167]; the metalogical framework proposal of Basin and Constable [8], who advocate constructive type theory as the framework logic; and the work of Matthews, Smaill, and Basin [111], who use Feferman's FS_0 [42], a logic designed with the explicit purpose of being a metalogical framework. A key advantage over the constructive type theory proposals is the greater simplicity and representational flexibility of rewriting logic as a framework logic. In practice, this translates into logic representations that are virtually isomorphic to the original logics that they represent and are therefore much easier to understand than complex encodings. In contrast, logical frameworks that—as most of those mentioned above—are based on higher-order type theories, although having very strong proof-theoretic power, have the disadvantage of often requiring complex translations or even lacking adequate representations for certain logics.

Our technical approach to the module calculus is related to the module composition ideas of Goguen and Burstall based on institutions [57] further developed in [9, 174, 36, 46], and to the encouraging experience that already exists in this regard in specification languages such as Clear [20], OBJ [66], ACT [39], Eqlog [61], and Maude [121]. More generally speaking, it is also related to other formalism- or language-dependent approaches. The approach of Bracha and Lindstrom [19] in the context of object-oriented languages using the lambda calculus as underlying formalism, and the metaobject-protocol approach [84] are two good examples among a variety of other proposals in the object-oriented programming area; the ML language is a fine example in the field of functional programming [71]. There is also an extensive body of work on applying higher-order lambda calculi with dependent [100] and/or polymorphic types [52, 170] to the design of highly modular functional languages, and to the semantics of inheritance in object-oriented languages. Our approach to modules shares with the institution-like approaches the advantage of being formalism-independent. The main difference is the openness and extensibility that reflection provides. The alternative in almost all other approaches is to offer a *fixed* repertoire of module-building operations, which essentially remain inaccessible and unmodifiable from the object level. By contrast, we propose an extensible module algebra with powerful and potentially unlimited possibilities of extension by new user-defined module operations.

Our proposed approach to reflection is related to a vast literature on reflection in programming languages [180, 178, 193, 190, 84, 75, 114], in theorem proving [201, 18, 176, 79, 34, 112, 56, 73], in concurrent and distributed computation [110, 162, 171], and in many other areas such as compilation, programming environments, operating systems, fault tolerance, and databases (see [177, 83] for recent snapshots of research in reflection). However, since the semantic foundations of reflection are not yet well understood, a distinctive advantage of our approach over previous approaches is that it is based on a general axiomatic notion of reflective logic [32] that, being part of the theory of general logics, is logic-independent. This advantage is combined with the efficient implementation of reflection in the Maude system, that supports a high-level declarative style of reflective programming.

Our proposed approach to constructing and interoperating reasoning and other tools is related to the many existing proof systems that are in fact a composition of multiple reasoning modules and to the numerous experiments in augmenting and combining existing proof systems. A detailed survey and discussion can be found in [54]. We give here a brief summary. The Boyer-Moore prover NQTHM [14, 16] (and the more recent ACL2 [82]) combines techniques for propositional reasoning, equality reasoning, typeset inference, term rewriting, and linear arithmetic. RRL [80, 81] includes a completion procedure, rewriting, associative-commutative theories, linear arithmetic, and Groebner basis over boolean ring. The KADS system [181] uses resolution augmented with special-purpose deciders. EHDM [189] and PVS [163] use a variety of ground decision procedures combined with rules for interactive proving. The Stanford Temporal Prover, STeP [41], combines model checking with deductive methods including contextual rewriting, generation of invariants and verification conditions, decision procedures for Presburger arithmetic, propositional and propositional temporal formulas, and a semi-decision procedure for first order logic. The experience of integrating a linear arithmetic module into NQTHM is described in detail in [17]. Rajan, Shankar and Srivas [169] describe the integration of a model checker with PVS. Harrison and Théry [72], and Ballarin, Homann and Calmet [7] report experiments of integration of the computer algebra system Maple [21] with HOL and Isabelle [166], respectively.

These experiments clearly demonstrate the need for composite, heterogeneous systems and the ability to both tightly integrate and loosely interoperate reasoning tools. However they have been carried out on a case-by-case basis. Our approach provides an infrastructure and methodology for integrating and interoperating reasoning tools with both other reasoning tools and tools of other sorts—symbolic evaluators, analysis tools, computer algebra tools— in a principled and sound manner.

H. Key Personnel

The key senior personnel are Dr. José Meseguer and Dr. Carolyn Talcott. Professor Peter Mosses of Aarhus University will participate during the first year of the project. Drs. Meseguer and Talcott will each dedicate 30% of their time to this research effort. Meseguer and Talcott are both funded by DARPA, ONR, and NSF.

Dr. Meseguer is a Principal Scientist in the Computer Science Lab at SRI where he leads the Declarative Languages and Architecture Group. Dr. Meseguer is the designer of the Maude language, and a co-designer of the OBJ, Eqlog, and FOOPS languages and of the Rewrite Rule Machine. He is especially well-known for his contributions to computer security, to models of concurrent computation, to the design and semantics of functional, logic and object-oriented programming languages, to software specification and formal methods in software engineering, to the theories of order-sorted algebra, order-sorted unification, abstract data types, and general logics, and to higher-order type theory. His latest research interests include, design and parallel implementation of declarative multi-paradigm programming languages, models of concurrent computation, software modularity and reusability, formal methods, logical frameworks, and massively concurrent architectures. Before joining SRI, he held a tenured academic position in Spain, and research positions at UCLA and UC Berkeley. He is associate editor of the journal *Theory and Practice of Object Systems* and has been a program committee member in a wide variety of technical conferences. Dr. Meseguer has given many invited lectures in international conferences and has published more than ninety technical papers.

Dr. Talcott is a Senior Research Associate in the Formal Reasoning Group of the Stanford University Computer Science Department. She is co-Editor-in-Chief of the *Journal of Lisp and Symbolic Computation.* Methods she has developed for reasoning about imperative, functional and concurrent programs are now being used by several research groups in the US, Europe, and Australia. She has given many invited lectures at universities, research laboratories, and workshops on this work. Research students from other universities frequently visit her group for periods ranging from a month to a year or more to work with her as part of their PhD research and she has served as external examiner for a number of PhD students. She has published papers in several related areas, including formalization and mechanized reasoning, composable semantic models for computation, logics for reasoning about programs, verification of a distributed garbage collection algorithm, and reasoning about dynamically adaptable runtime environments.

Professor Mosses is an Associate Professor at the Department of Computer Science, University of Aarhus, Denmark, and a Senior Researcher at BRICS, Centre for Basic Research in Computer Science at the Universities of Aarhus and Aalborg. As a student of Strachey, and later Junior Research Fellow, at Oxford, he was involved in the development of denotational semantics, and pioneered partial evaluation in a semantics prototyping system, SIS. His dissatisfaction with the modularity of denotational semantics led him to the development of action semantics, combining ideas of denotational, operational, and algebraic semantics in a single framework and enabling the rigorous application of formal semantics to large-scale practical programming languages. He has published the definitive book on action semantics, and given several tutorials on it at international conferences. He has also developed the original framework of unified algebras, for specification of abstract data types involving type constructors and type inclusions. Since 1995 he has been overall coordinator of CoFI (the Common Framework Initiative for algebraic specification and development) and a major contributor to its design of the Common algebraic specification language CASL. He has recently been appointed chairman of IFIP WG 1.3 (Foundations of System Specification) and he is a member of IFIP WG 2.2 (Formal Description of Programming Concepts). He has been a Senior Visiting Research Fellow at Edinburgh, and Head of Department at Aarhus. He was the organizer of TAPSOFT'95, and has served as a PC member in many international conferences. He has given invited lectures on both action semantics and algebraic specifications at international conferences. His research interests include also concurrency.

Previous Accomplishments

Drs. Meseguer and Talcott bring a broad range of accomplishments and expertise to the project. Currently, they are collaborating on several projects related to formal interoperability and semantic foundations for composition in open systems. These project combine the work on General Logics, mappings, and Rewriting Logic with the work on Open Mechanized Reasoning Systems (OMRS) and on Actor Semantics. The Rewriting Logic/OMRS work provides a formal semantic framework for describing diverse aspects of a system and the contexts in which it exists. The Rewriting Logic/Actors work establishes a formal framework for specifying and composing components of open systems and provides challenges in the development of Maude to support open object-based distributed computation and computational reflection.

José Meseguer

Dr. Meseguer's relevant accomplishments span the following areas:

1. *Logic and Formal Methods*, including: Formal Models of Security [59, 60]; Type Theory [118, 179, 125, 64, 124, 91, 92]; Linear, Rewriting, and Equational Logics [94, 93, 95, 119, 98, 96, 64, 124, 123]; Logics and Models of Concurrency [127, 35, 23, 95, 119, 128, 130, 175, 129]; Logical Frameworks (General Logics) [117, 120, 24, 96, 97, 126]; Formal Methods, Specification and Verification, and Parameterized Programming Methodology [96, 119, 65, 50, 66, 121]

2. *Declarative Specification and Programming Languages*, including: Equational Languages (OBJ) [65, 49, 85, 66]; Relational and Equational Languages (Eqlog) [61, 62]; Multiparadigm Concurrent Object-Oriented Languages (FOOPS and Maude) [63, 132, 121, 120, 122, 131, 89]; and Reflective Languages [32, 30].

In 1990 Dr. Meseguer proposed rewriting logic as a new specification and programming formalism for concurrent and distributed systems. Since that time, more than sixty papers developing the rewriting logic ideas have been published by an active research community in the US, Europe and Japan, three rewriting logic-based languages have been implemented, and a first international meeting on rewriting logic has been held. He is the original designer of the Maude language and has led its current prototype implementation. He has also researched in depth the applications of rewriting logic and Maude to object-oriented specification and programming and to real-time systems. With other members of SRI's Declarative Languages and Architecture Group he has designed, prototyped and evaluated a new rewriting-logic-based massively parallel architecture called the Rewrite Rule Machine and its compilation tools and techniques.

OBJ and Maude

Since 1980 Dr. Meseguer has worked together with Joseph Goguen and other coworkers in the design, formal semantics, and implementation of new, more powerful, versions of the equational language OBJ [65, 49, 58, 66]. There is a growing international community of OBJ users, including about 180 universities and research laboratories to whom we have distributed the OBJ3 system.

Maude [121] is a wide spectrum language based on rewriting logic that contains OBJ as a functional sublanguage and supports executable specification and programming of concurrent object-oriented systems and of general concurrent systems. Maude can also be used as a *logical framework* in which many different logics and languages can be specified and prototyped [96]. With Manuel Clavel, and Drs. Steven Eker and Patrick Lincoln we have also developed a prototype sequential Maude implementation written in C++ [28] that combines the flexibility of an interpreter with the competitive speed derived from advanced semicompilation techniques (590,000 rewrites per second on a Pentium PC). Studies of parallel compilation techniques [132, 90] indicate that a subset of Maude called Simple Maude could be used as a machine-independent parallel programming language and could be implemented with reasonable efficiency on MIMD, SIMD, and MIMD/SIMD machines.

Carolyn Talcott

Dr. Talcott has substantial expertise in formal reasoning and in developing semantics and methods for reasoning about higher-order programs with effects, both sequential and concurrent. In [103, 104, 105] a language, λ_{mk}, based on the call-by-value lambda calculus extended by reference primitives à la ML was studied: methods for defining operational semantics and for proving program equivalence were developed and a complete axiomatization of equivalence in the zero-order fragment of λ_{mk} was given. In [76, 77] VTLoE, a variable typed logic of effects for specification and reasoning about λ_{mk} programs was introduced. VTLoE has been used for reasoning about object-based programs [106] and a first-order fragment has been implemented in Isabelle [48]. In [183] Dr. Talcott proposed a general framework for defining imperative functional languages with associated methods for establishing program equivalence. This generalizes the results for λ_{mk} to a wide class of higher-order languages with effects. The ideas and methods developed in this work have been used to prove correctness of a number of nontrivial program transformations and compiler optimizations, both for imperative functional languages.

Dr. Talcott in collaboration with Professors Agha, Mason and Smith have developed a formal theory of components of open distributed systems based on the actor model of computation [74, 1, 2]. The formal semantics of a higher-order actor language is presented in [3, 4] and a rich equational theory for actor programs was developed. This provides a basis for establishing correctness of compiler optimizations and other operations on actor programs. In [187] two important notions are introduced: *actor theories* (called *abstract actor structures* in early papers) and *interaction semantics*. Actor theories provide a framework for high-level, programming language independent, specification of individual actor behaviors. Interaction semantics abstracts away from the details of internal computation and provides a means of reasoning about actor system components considered as black boxes. Actor theories are formalized as in rewriting logic in [186] and they are used as the starting point of a specification formalism [37]. In [107] a semantics preserving translation between two actor languages is presented. The proof of correctness is of interest since it demonstrates a methodology based on actor theories for proving correctness of transformations and translations of actor languages and more generally for reasoning about components of open distributed systems. Interaction semantics is endowed with an algebra structure that provides a basis for modular reasoning about construction and transformation of components specified as abstract actor systems [188].

The OMRS project [55, 53] is a joint project with the Mechanized Reasoning Group of IRST in Trento, Italy. The ultimate goal of this work is to provide a framework and a methodology for constructing and modifying complex reasoning systems by composing and integrating simpler modules. The OMRS framework has been used to give a specification of parts of the Boyer-Moore prover, NQTHM, focusing on the integration of the linear-arithmetic module as described in [15].

Peter Mosses

Professor Mosses is an Associate Professor at the Department of Computer Science, University of Aarhus, Denmark, and a Senior Researcher at BRICS, Centre for Basic Research in Computer Science at the Universities of Aarhus and Aalborg. His relevant accomplishments are in the areas of programming language semantics, algebraic specification, and development of prototyping tools.

Prof. Mosses was involved in the development of denotational semantics [135], and pioneered partial evaluation in a semantics prototyping system, SIS [137, 136]. He is the author of a Handbook chapter on denotational semantics [145], whose systematic use of domain constructors inspired Moggi to propose the use of monads [133].

Directly adopting ideas from algebraic specification, Prof. Mosses has developed a framework of abstract semantic algebras [138, 139, 140, 141]. In collaboration with Dr. D. A. Watt, he has established the framework of action semantics, which has both algebraic and operational foundations and introduces a novel style of formal notation [158, 146, 148, 147, 198, 151, 155, 154]. Action semantics of concurrency constructs have been developed [147, 149, 157]. With Dr. A. van Deursen he has created translation and checking tools for action semantics [191]. Action semantics has been used to give formal semantics of practical programming languages such as Pascal [159] and Standard ML [197, 200]. It has been used in an industrial context for the formal semantics of AND-F [161, 68], the Architecture Neutral Distribution Format developed by the Open Software Foundation.

[158, 146, 148, 147, 198, 159, 155, 154]. Action semantics of concurrency constructs have been developed [147, 149, 157]. With Dr. A. van Deursen he has created translation and checking tools for action semantics [191].

Mosses has developed the novel specification framework of unified algebras [144, 142, 143, 150, 152] specifically to facilitate datatype specification in action semantics, where it is desirable to be able to use type union and other type constructors in an algebraic setting.

Since 1995, Mosses has been overall coordinator of CoFI (the Common Framework Initiative for algebraic specification and development) and a major contributor to its design of the Common algebraic specification language CASL [153, 156].

Section III. Additional Information

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. Towards a theory of actor computation. In *The Third International Conference on Concurrency Theory (CONCUR '92)*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer Verlag, August 1992.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[5] E. Astesiano and M. Cerioli. Relationships between logical frameworks. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, volume 655 of *LNCS*, pages 126–143. Springer-Verlag, 1993.

[6] A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, December 1992.

[7] C. Ballarin, K. Homann, and J. Calmet. Theorems and Algorithms: an Interface between Isabelle and Maple. In *Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC'95)*, pages 150–157. ACM PRESS, 1995.

[8] D. A. Basin and R. L. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.

[9] J.A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37:335–372, 1990.

[10] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

[11] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[12] Robert H. Bourdeau and Betty H. C. Cheng. An object-oriented toolkit for constructing specification editors. In *Proceedings of COMPSAC'92: Computer Software and Applications Conference*, pages 239–244, 1992.

[13] Robert H. Bourdeau and Betty H. C. Cheng. A formal semantics of object models. *IEEE Trans. on Software Engineering*, 21(10):799–821, October 1995.

[14] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[15] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. Technical Report ICSCA-CMP-44 , Institute for Computing Science, University of Texas at Austin, 1985.

[16] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[17] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence 11*. Oxford University Press, 1988.

[18] R.S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 103–185. Academic Press, 1981.

[19] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the International Conference on Computer Languages*, pages 282–290. IEEE, 1992.

[20] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.

[21] Char B.W., Geddes K.O., Gonnet G.H., Leong B.L., Monagan M.B., and Watt S.M. *Maple V Language Reference Manual*. Springer–Verlag, 1992.

[22] Manuel Calvel, Francisco Durán, Steven Eker, and José Meseguer. Design and implementation of the cafe prover and church-rosser checker tools. Technical report, SRI International, December 1997.

[23] Ross Casley, Roger Crew, José Meseguer, and Vaughan Pratt. Temporal structures. *Mathematical Structures in Computer Science*, 1(2):179–213, 1991.

[24] Maura Cerioli and José Meseguer. May I borrow your logic? In *Proceedings of MFCS'93, 18th International Symposium on Mathematical Foundations of Computer Science*, pages 342–351. Springer LNCS 711, 1993.

[25] Betty H. C. Cheng, Enoch Y. Wang, and Robert H. Bourdeau. A graphical environment for formally developing object-oriented software. In *Proc. of IEEE 6th International Conference on Tools with Artificial Intelligence*, 1994.

[26] Betty H.C. Cheng, Enoch Y. Wang, Robert H. Bourdeau, and Heather A. Richter. Bridging the gap between the informal and formal approaches to software development. In *Proc. of Software Engineering Research Forum*, November 1995.

[27] L. J. M. Claesen and M. J. C. Gordon, editors. *Higher Order Logic Theorem Proving and its Applications*. Elsevier Science Publishers B. V. (North-Holland), 1993.

[28] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Rewriting Logic Workshop'96*, 1996.

[29] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Reflection'96*, pages 263–288. Xerox PARC, 1996.

[30] M. Clavel and J. Meseguer. Reflection in rewriting logic. In *Rewriting Logic Workshop'96*, 1996.

[31] Manuel Clavel. Reflection in general logics, rewriting logic, and Maude. Ph.D. Thesis, submitted to the University of Navarre, 1997.

[32] Manuel G. Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

[33] A. Coglio, F. Giunchiglia, P. Pecchiari, and C. Talcott. A logic level specification of the nqthm simplification process. Technical report, IRST, University of Genova, Stanford University, 1997.

[34] R. L. Constable. Metalogical frameworks II: Using reflected decision procedures. Lecture at the Max Planck Institut für Informatik, Saarbrücken, Germany, July 1993.

[35] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing the algebra of net computations and processes. Technical Report SRI-CSL-90-12, SRI International, Computer Science Laboratory, November 1990. To appear in *Acta Informatica*.

[36] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. To appear, *Proceedings* of Workshop on Logical Frameworks (Edinburgh, Scotland, May 1991), 1992.

[37] C. H. C Duarte. Towards a proof-theoretic foundation for actor specification and verification, 1996. submitted.

[38] H. Ehrig, M. Baldamus, and F. Orejas. New concepts of amalgamation and extension of a general theory of specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, volume 655 of *LNCS*, pages 199–221. Springer-Verlag, 1993.

[39] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[40] David Espinosa. *Semantic Lego.* PhD thesis, Columbia University, 1995. WWW[2].

[41] Zohar Manna et. al. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford Computer Science Department, June 1994.

[42] S. Feferman. Finitary inductively presented logics. In R. Ferro et al., editors, *Logic Colloquium'88*, pages 191–220. North-Holland, 1989.

[43] A. Felty and D. Miller. Encoding a dependent-type λ-calculus in a logic programming language. In M.E. Stickel, editor, *Proc. 10th. Int. Conf. on Automated Deduction, Kaiserslautern, Germany, July 1990*, volume 449 of *LNCS*, pages 221–235. Springer-Verlag, 1990.

[44] J. Fiadeiro and T. Maibaum. Generalising interpretations between theories in the context of (π-)institutions. In G. Burn, S. Gay, and M. Ryan, editors, *Theory and Formal Methods 93*, pages 126–147. Springer-Verlag, 1993.

[2]http://www-swiss.ai.mit.edu/ftpdir/users/dae/home.html

[45] J. Fiadeiro and A. Sernadas. Structuring theories on consequence. In D. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, pages 44–72. Springer LNCS 332, 1988.

[46] J.L. Fiadeiro and T. Maibaum. Interconnecting formalisms: supporting modularity, reuse and incrementality. In *Proceedings of the 1995 SIGSOFT Conference, Washington DC, USA*, pages 72–80. ACM, 1995.

[47] Andrzej Filinski. *Controlling Effects.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[48] J. Frost. *Effective Programming.* PhD thesis, Technical University of Denmark, 1996. Available as IT-TR-1996-001.

[49] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[50] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming and its application to rapid prototyping in OBJ2. In Y. Matsumoto and Y. Ohno, editors, *Japanese Perspectives in Software Engineering*, pages 77–102. Addison-Wesley, 1989.

[51] P. Gardner. *Representing Logics in Type Theory.* PhD thesis, Technical Report CST-93-92, Department of Computer Science, University of Edinburgh, 1992.

[52] Jean-Yves Girard. *Interprétation Fonctionelle et Élimination des Coupures dans l'Arithmétique d'ordre Supérieure.* PhD thesis, Univ. Paris VII, 1972.

[53] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems. In *Workshop on Frontiers of Combining Systems FROCOS'96*, 1996.

[54] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems, 1996. submitted for publication.

[55] F. Giunchiglia, P. Pecchiari, and C. L. Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems. Technical Report 9409-15, IRST, November 1994. Also appears as Stanford University Computer Science Department Technical Note STAN-CS-94-TN-15.

[56] Fausto Giunchiglia, Paolo Traverso, Alessandro Cimatti, and Paolo Pecchiari. A system for multi-level reasoning. In *IMSA'92*, pages 190–195. Information-Technology Promotion Agency, Japan, 1992.

[57] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[58] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy Winkler. An introduction to OBJ3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting, Orsay, France, July 8-10, 1987*, pages 258–263. Springer LNCS 308, 1988.

[59] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[60] Joseph Goguen and José Meseguer. Unwinding and inference control. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 75–86. IEEE, 1984.

[61] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.

[62] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22. Springer-Verlag, 1987. LNCS, Volume 250; extended version to appear in *J. Logic Programming.*

[63] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[64] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. Originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; several draft and technical report versions were circulated since 1985.

[65] Joseph Goguen, José Meseguer, and David Plaisted. Programming with parameterized abstract objects in OBJ. In Domenico Ferrari, Mario Bolognani, and Joseph Goguen, editors, *Theory and Practice of Software Technology*, pages 163–193. North-Holland, 1983.

[66] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1996. To appear in J.A. Goguen, editor, *Applications of Algebraic Specification Using OBJ*, Cambridge University Press.

[67] Bo Stig Hansen and Jørgen Bundgaard. The role of the ANDF formal specification. Technical Report 202104/RPT/5, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1992.

[68] Bo Stig Hansen and Jens Ulrik Toft. The formal specification of ANDF, an application of action semantics. In [151], pages 34–42, 1994.

[69] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[70] R. Harper, D. Sannella, and A. Tarlecki. Structure theory presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[71] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Dept. of Computer Science, University of Edinburgh, 1986.

[72] J. Harrison and L. Théry. Extending the HOL Theorem Prover with a Computer Algebra System to Reason about the Reals. In Claesen and Gordon [27], pages 174–184.

[73] John Harrison. Metatheory and reflection in theorem proving: a survey and critique. University of Cambridge Computer Laboratory, 1995.

[74] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[75] Patricia Hill and John Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[76] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A theory of classes for a functional language with effects. In *Proceedings of CSL92*, volume 702 of *Lecture Notes in Computer Science*, pages 309–326. Springer, Berlin, 1993.

[77] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 1995.

[78] D.J. Howe. Semantic foundations for embedding HOL in Nuprl. Manuscript; Bell Labs, 1996.

[79] Douglas J. Howe. Reflecting the semantics of reflected proof. In Peter Aczel, Harold Simmons, and Stanley S. Wainer, editors, *Proof Theory*, pages 229–250. Cambridge University Press, 1990.

[80] D. Kapur and H. Zhang. First-order theorem proving using conditional rewrite rules. In *9th International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 1–20, 1988.

[81] D. Kapur and H. Zhang. An overview of RRL (rewrite rule laboratory). In *Third International Conf. of Rewriting Techniques and Applications*, 1989.

[82] M. Kaufmann and J S. Moore. Design Goals for ACL2. Technical Report 101, Computational Logic Inc., Austin, Texas, 1994.

[83] G. Kiczales, editor. *Reflection'96*, 1996.

[84] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[85] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, editors, *Proceedings, 15th Intl. Coll. on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer LNCS 317, 1988.

[86] Michael R. Laux, Robert H. Bourdeau, and Betty H. C. Cheng. An integrated development environment for formal specifications. In *Proc. of International Conference on Software Engineering and Knowledge Engineering*, pages 681–688, 1993.

[87] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996. WWW[3].

[3] http://www-swiss.ai.mit.edu/ftpdir/users/dae/home.html

[88] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL'95, 22nd ACM Symposium on Principles of Programming Languages*, pages 333–343. ACM, 1995. WWW[4].

[89] Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.

[90] Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Livio Ricciulli. Compiling rewriting onto SIMD and MIMD/SIMD machines. In *Proceedings of PARLE'94, 6th International Conference on Parallel Architectures and Languages Europe*, pages 37–48. Springer LNCS 817, 1994.

[91] Narciso Martí-Oliet and José Meseguer. Inclusions and subtypes I: First order case. To appear in *J. Logic and Computation*; appeared also as SRI Report SRI-CSL-90-16.

[92] Narciso Martí-Oliet and José Meseguer. Inclusions and subtypes II: Higher-order case. To appear in *J. Logic and Computation*; appeared also as SRI Report SRI-CSL-90-16.

[93] Narciso Martí-Oliet and José Meseguer. An algebraic axiomatization of linear logic models. In G.M. Reed, A.W. Roscoe, and R. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 335–355. Oxford University Press, 1991.

[94] Narciso Martí-Oliet and José Meseguer. From Petri nets to linear logic. *Math. Struct. in Comp. Sci.*, 1:69–101, 1991.

[95] Narciso Martí-Oliet and José Meseguer. From Petri nets to linear logic through categories: a survey. *Intl. J. of Foundations of Comp. Sci.*, 2(4):297–399, 1991.

[96] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[97] Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.

[98] Narciso Martí-Oliet and José Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhoefer, editors, *Theoretical Approaches to Dynamic Worlds in Artificial Intelligence and Computer Science*. 1996. To be published by Kluwer Academic Publisher.

[99] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.

[100] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[4] http://www-swiss.ai.mit.edu/ftpdir/users/dae/home.html

[101] I. A. Mason. *The Semantics of Destructive Lisp.* PhD thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

[102] I. A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10:177–210, 1988.

[103] I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, volume 372 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 1989.

[104] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[105] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.

[106] I. A. Mason and C. L. Talcott. Reasoning about object systems in vtloe. *International Journal of Foundations of Computer Science*, 6(3):265–298, September 1995.

[107] I. A. Mason and C. L. Talcott. A semantically sound actor translation. In *ICALP'97*, number 1256 in Lecture Notes in Computer Science, pages 369–378, 1997.

[108] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages. In *Proceedings of OOPSLA'95*, pages 300–315, 1995. SIGPLAN Notices Vol 30, No. 10.

[109] H. Masuhara and A. Yonezawa. Reasoning-conscious meta-object design of a reflective concurrent language, 1997. manuscript, Sept 97.

[110] Satoshi Matsuoka, Takuo Watanabe, Yuuji Ichisugi, and Akinori Yonezawa. Object-oriented concurrent reflective architectures. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, pages 211–226. Springer LNCS 612, 1992.

[111] S. Matthews, A. Smaill, and D. Basin. Experience with FS_0 as a framework theory. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.

[112] Seán Matthews. Reflection in logical systems. In *IMSA'92*, pages 178–183. Information-Technology Promotion Agency, Japan, 1992.

[113] Brian H. Mayoh. Galleries and institutions. Technical Report DAIMI PB-191, Computer Science Dept., Aarhus University, 1985.

[114] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.

[115] J. Meseguer and C. Talcott. Reasoning theories and rewriting logic. (in preparation).

[116] J. Meseguer and C. Talcott. Semantic interoperation of dynamic heterogeneous architectures, 1997. Technical presentations to EDCS Architecture Cluster Meeting, April and July 1997.

[117] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[118] José Meseguer. Relating Models of Polymorphism. In *Proc. POPL'89*, pages 228–241. ACM, 1989.

[119] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[120] José Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proc. 3rd Intl. Conf. on Algebraic and Logic Programming*, pages 158–200. Springer LNCS 632, 1992.

[121] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[122] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.

[123] José Meseguer. Membership algebra as a semantic framework for equational specification. To appear in F. Parisi-Presicce, ed., Proc. WADT'97, Springer LNCS, 1998.

[124] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, 1993.

[125] José Meseguer, Joseph Goguen, and Gert Smolka. Order-sorted unification. *J. Symbolic Computation*, 8:383–413, 1989.

[126] José Meseguer and Narciso Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, Santa Margherita, Italy, May/June 1994*, pages 48–80. Springer LNCS 906, 1995.

[127] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.

[128] José Meseguer, Ugo Montanari, and Vladimiro Sassone. On the semantics of Petri nets. In W.R. Cleaveland, editor, *Proceedings of the Concur'92 Conference, Stony Brook, New York, August 1992*, pages 286–301. Springer LNCS 630, 1992.

[129] José Meseguer, Ugo Montanari, and Vladimiro Sassone. On the model of computation of Place/Transition Petri nets. In *Proceedings 15th International Conference on Application and Theory of Petri Nets*, pages 16–38. Springer LNCS 815, 1994.

[130] José Meseguer, Ugo Montanari, and Vladimiro Sassone. Process versus unfolding semantics for Place/Transition Petri nets. *Theoretical Computer Science*, 153(1–2):171–210, 1996.

[131] José Meseguer and Xiaolei Qian. A logical semantics for object-oriented databases. In *Proc. International SIGMOD Conference on Management of Data*, pages 89–98. ACM, 1993.

[132] José Meseguer and Timothy Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.

[133] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.

[134] Eugenio Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 138–139. Springer-Verlag, 1991. Invited talk.

[135] Peter D. Mosses. The mathematical semantics of Algol60. Tech. Mono. PRG–12, Programming Research Group, Univ. of Oxford, 1974.

[136] Peter D. Mosses. Compiler generation using denotational semantics. In *MFCS'76, Proc. Symp. on Math. Foundations of Computer Science, Gdańsk*, volume 45 of *Lecture Notes in Computer Science*. Springer-Verlag, 1976.

[137] Peter D. Mosses. SIS, Semantics Implementation System: Reference manual and user guide. Tech. Mono. MD–30, Dept. of Computer Science, Univ. of Aarhus, 1979. Out of print.

[138] Peter D. Mosses. A constructive approach to compiler correctness. In *ICALP'80, Proc. Int. Coll. on Automata, Languages, and Programming, Noordwijkerhout*, volume 85 of *Lecture Notes in Computer Science*, pages 449–469. Springer-Verlag, 1980.

[139] Peter D. Mosses. A semantic algebra for binding constructs. In *Proc. Int. Coll. on Formalization of Programming Concepts, Peñiscola*, volume 107 of *Lecture Notes in Computer Science*, pages 408–418. Springer-Verlag, 1981.

[140] Peter D. Mosses. Abstract semantic algebras! In *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*, pages 45–71. North-Holland, 1983.

[141] Peter D. Mosses. A basic abstract semantic algebra. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis*, volume 173 of *Lecture Notes in Computer Science*, pages 87–107. Springer-Verlag, 1984.

[142] Peter D. Mosses. Unified algebras and action semantics. In *STACS'89, Proc. Symp. on Theoretical Aspects of Computer Science, Paderborn*, volume 349 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[143] Peter D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989.

[144] Peter D. Mosses. Unified algebras and modules. In *POPL'89, Proc. 16th Ann. ACM Symp. on Principles of Programming Languages*, pages 329–343. ACM, 1989.

[145] Peter D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[146] Peter D. Mosses. A practical introduction to denotational semantics. In *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 1–49. Springer-Verlag, 1991.

[147] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[148] Peter D. Mosses. An introduction to action semantics. In *Logic and Algebra of Specification, Proc. Marktoberdorf Summer School 1991*, volume 94 of *NATO ASI Series F*, pages 247–288. Springer-Verlag, 1993.

[149] Peter D. Mosses. On the action semantics of concurrent programming languages. In *Semantics: Foundations and Applications, Proc. REX Workshop, Beekbergen, 1992*, volume 666 of *Lecture Notes in Computer Science*, pages 398–424. Springer-Verlag, 1993.

[150] Peter D. Mosses. The use of sorts in algebraic specifications. In Michel Bidoit and Christine Choppy, editors, *Recent Trends in Data Type Specification, Proc. 8th Workshop on Specification of Abstract Data Types, Dourdan, France, 1991, Selected Papers*, volume 655 of *Lecture Notes in Computer Science*, pages 66–91. Springer-Verlag, 1993.

[151] Peter D. Mosses, editor. *Proc. 1st Intl. Workshop on Action Semantics, Edinburgh, 1994*, number NS-94-1 in BRICS Notes Series. BRICS, Dept. of Computer Science, Univ. of Aarhus, 1994. WWW[5].

[152] Peter D. Mosses. Unified algebras and abstract syntax. In *Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, 1992, Selected Papers*, volume 785 of *Lecture Notes in Computer Science*, pages 280–294. Springer-Verlag, 1994.

[153] Peter D. Mosses, editor. *CoFI: Initiative for a Common Framework for Algebraic Specification*, URL: http://www.brics.dk/Projects/CoFI, 1996.

[154] Peter D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996. WWW[6], FTP[7].

[155] Peter D. Mosses. A tutorial on action semantics. 50pp. Tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996). WWW[8], FTP[9], March 1996.

[5]http://www.brics.dk/NS/94/1/BRICS-NS-94-1/BRICS-NS-94-1.html
[6]http://www.brics.dk/RS/96/53/BRICS-RS-96-53/BRICS-RS-96-53.html
[7]ftp.brics.dk/Projects/AS/Papers/Mosses96MFCS
[8]http://www.brics.dk/NS/96/14/BRICS-NS-96-14/BRICS-NS-96-14.html
[9]ftp.brics.dk/Projects/AS/Papers/Mosses96DRAFT

[156] Peter D. Mosses. CoFI: The Common Framework Initiative for algebraic specification and development. In *TAPSOFT'97, Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 115–137. Springer-Verlag, 1997. Invited lecture.

[157] Peter D. Mosses and Martín A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *Lecture Notes in Computer Science*, pages 461–479. Springer-Verlag, 1994. WWW[10].

[158] Peter D. Mosses and David A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.

[159] Peter D. Mosses and David A. Watt. Pascal action semantics, version 0.6. FTP[11], March 1993.

[160] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth Int. Joint Conf. and Symp. on Logic Programming*, pages 810–827. The MIT Press, 1988.

[161] Jens P. Nielsen and Jens Ulrik Toft. Formal specification of ANDF, existing subset. Technical Report 202104/RPT/19, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1994.

[162] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *IMSA'92*, pages 36–47. Information-Technology Promotion Agency, Japan, 1992.

[163] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction–CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

[164] Y. Oyama, K. Taura, and A. Yonezawa. An efficient compilation framework for languages based on concurrent process calculi. In *Proceedings of Euro-Par'97 Object-Oriented Programming*. Springer, 1997.

[165] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–39, 1989.

[166] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[167] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proc. Fourth Annual IEEE Symp. on Logic in Computer Science*, pages 313–322, Asilomar, California, June 1989.

[10] http://www.brics.dk/RS/94/20/BRICS-RS-94-20/BRICS-RS-94-20.html

[11] ftp.brics.dk/Projects/AS/Papers/MossesWatt93DRAFT

[168] A. Poigné. Foundations are rich institutions, but institutions are poor foundations. In H. Ehrig et al., editors, *Categorical Methods in Computer Science with Aspects from Topology*, volume 393 of *LNCS*, pages 82–101. Springer-Verlag, 1989.

[169] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97. Springer-Verlag, June 1995.

[170] J.C. Reynolds. Towards a Theory of Type Structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425. Springer LNCS 19, 1974.

[171] Luis H. Rodriguez, Jr. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In *IMSA'92*, pages 107–112. Information-Technology Promotion Agency, Japan, 1992.

[172] A. Salibra and G. Scollo. Compactness and Löwenheim-Skolem properties in pre-institution categories. Technical Report LIENS-92-10, Laboratoire d'Informatique de l'Ecole Normale Supérieure, Paris, March 1992.

[173] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[174] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[175] Vladimiro Sassone, José Meseguer, and Ugo Montanari. Inductive completion of monoidal categories and infinite net computations. Submitted for publication.

[176] N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge University Press, 1994.

[177] Brian Smith and Akinori Yonezawa, editors. *Proc. of the IMSA'92 International Workshop on Reflection and Meta-Level Architecture, Tokyo, November 1992.* Research Institute of Software Engineering, 1992.

[178] Brian C. Smith. Reflection and Semantics in Lisp. In *Proc. POPL'84*, pages 23–35. ACM, 1984.

[179] Gert Smolka, Werner Nutt, Joseph Goguen, and José Meseguer. Order-sorted equational computation. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 297–367. Academic Press, 1989.

[180] G.L Steele, Jr. and G.J. Sussman. The art of the interpreter or, the modularity, complex, technical memorandum AIM-453. Technical report, MIT AI-Lab, May 1978.

[181] M. Stickel. KLAUS automated deduction system. In *Proceedings of the ninth conference on automated deduction*, number 310 in Lecture Notes in Computer Science, pages 750–751. Springer-Verlag, 1988.

[182] I. Sutherland and R. Platek. A plea for logical infrastructure. In *TTCP XTP-1 Workshop on Effective Use of Automated Reasoning Technology in System Development*, pages 1–3, 1992.

[183] C. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1996.

[184] C. L. Talcott. *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation*. PhD thesis, Stanford University, 1985.

[185] C. L. Talcott. Programming and proving with function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University Computer Science Department, 1989.

[186] C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, number 4 in Electronic Notes in Theoretical Computer Science, pages 360–383. North Holland, 1996.

[187] C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J-B. Stefani, editors, *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996. Proceedings published in 1997 by Chapman & Hall.

[188] C. L. Talcott. Composable semantic models for actor theories. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Science*, number 1281 in Lecture Notes in Computer Science, pages 321–364. Springer-Verlag, 1997.

[189] The SRI Staff. The SRI specification and verification system, users guide. Technical report, SRI International, 1986.

[190] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[191] Arie van Deursen and Peter D. Mosses. ASD: The action semantic description tools. In *AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology, Munich*, volume 1101 of *Lecture Notes in Computer Science*, pages 579–582. Springer-Verlag, 1996.

[192] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.

[193] M. Wand and D.P. Friedman. The mystery of the tower revealed. *Lisp and Symbolic Computation*, 1(1):11–38, 1988.

[194] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *Proc. of IEEE International Conference on Software Engineering (ICSE97)*, 1997.

[195] Keith Wansbrough. A modular monadic action semantics. Master's thesis, Dept. of Computer Science, Univ. of Auckland, February 1997. WWW[12].

[196] Keith Wansbrough and John Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997. WWW[13].

[12] http://www.dcs.gla.ac.uk/~keithw/thesis
[13] http://www.dcs.gla.ac.uk/~keithw/thesis

[197] David A. Watt. An action semantics of Standard ML. In *Proc. Third Workshop on Math. Foundations of Programming Language Semantics, New Orleans*, volume 298 of *Lecture Notes in Computer Science*, pages 572–598. Springer-Verlag, 1988.

[198] David A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

[199] David A. Watt. JOOS action semantics. FTP[14], October 1997.

[200] David A. Watt. Standard ML action semantics. FTP[15], May 1997.

[201] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[202] Martin Wirsing and Alexander Knapp. A formal approach to object oriented software engineering. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, number 4 in Electronic Notes in Theoretical Computer Science. North Holland, 1996.

[14] http://www.dcs.gla.ac.uk/~daw/research-projects/JAS.html
[15] http://www.dcs.gla.ac.uk/~daw/research-projects/AAS.html