Typing for Reliable Distributed Systems - Recent Advances

Paweł T. Wojciechowski
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
Pawel.Wojciechowski@epfl.ch

Abstract

In this paper, we report on recent progress in the area of language-based software dependability. We give examples of using advanced type systems for statically detecting non-trivial programming errors in programs. We have chosen examples that are particularly applicable to concurrent, distributed and modular (or component-based) systems.

Key-words: concurrency, distributed systems, provable dependability, software verification, type systems, modularity, components, language constructs and features

1 Introduction

The programming language research community traditionally focuses on designing programming abstractions that would improve the existing programming practice, enabling software developers to build high-quality code in a shorter time. Today's computer systems are predominantly concurrent and distributed. There has been recently a lot of interest in developing language-based tools and verification methods that could make such systems *provably dependable*. In this paper we describe some example work in this area, including our own work on safe declarative synchronization [40], and typing for isolation-only, rollback-free transactions [41].

The goal of this paper is to communicate some of the most recent theoretical and experimental results developed in the programming language community, to practitioners and researchers who are experienced in the design of dependable systems. We hope that our paper would be useful to set up future directions for both research communities. Due to limited space, we only focus on one aspect, i.e. developing advanced type systems [5, 24, 25] and typecheckers to detect some class of programming errors in concurrent and distributed systems.

Concurrency, distribution (on many network sites) and modularity (for code reuse, reconfiguration and dynamic adaptation) are the main characteristics of many dependable critical systems. Unfortunately, concurrency and distribution make programming of such systems tricky and error-prone. Errors in concurrent and distributed programs are hard to detect by testing. Moreover, since programs can be built from separately compiled modules or components, it may be impossible to modify the program, e.g. in order to perform different test experiments with different input values. More importantly, testing is time-consuming and does not guarantee that there are no errors.

A typechecker can *guarantee* that a program is free of certain classes of errors. Type-based verification is therefore an effective way to increase system reliability. The idea is that the programmers should spend some effort and annotate their programs with *types*, which are declarations or expressions built on top of the host language, say Java [3]. It is often possible to design algorithms for inferring the type annotations, so that the amount of necessary coding is minimized. The types would help the compiler to understand what the program is expected to do, and typecheck (at compile or link time) if certain desirable properties hold.

Previous research demonstrated the use of static typechecking for memory-access safety, e.g. to detect dangling pointers. More recent work includes verifying concurrency properties, such as freedom from race conditions and deadlock, as well as type-safe distributed programming, e.g. safe data marshalling and software version control to support interoperation of old and new code. Types also support building of reliable systems from *reusable components*: the module or component interfaces can be extended with dependent types that describe some aspects of a module's behaviour. This enables to detect some compositional errors without the need to look into the source code of components.

The paper has the following structure. We first motivate domain-specific languages and type-based verification for reliable distributed systems. Then, we explain what are the type systems, and describe some selected work on types for concurrent and distributed programming. Finally, we turn to safe composition of network components, and describe some existing work, including our own ongoing work. Finally, we conclude.

2 Motivation

Despite many research and engineering efforts in the software industry and at the universities, distributed systems are not fundamentally more reliable than they were a decade ago. The increasing dependence by industry and society on distributed systems means that *reliability* of such systems is crucial for normal functioning and safety in our world.

Distributed systems consist of processes that execute on separate machines and communicate using some (wired or wireless) network connections. Since machines may crash or be rebooted at any time, and the Internet connections are unreliable, distributed systems must deal with partial failure. The complexity of dealing with partial failure is usually hidden at the level of *network components*. The network components include standard network protocols such as TCP/IP, as well as a growing number of middleware protocols, such as *group communication* protocols [17].

Protocols are highly concurrent; testing can often be useful to find common errors but is not capable of guaranteeing correctness (*correctness* informally means compliance of program execution with our assumptions). There is therefore a natural demand for the most critical parts of protocols to be analyzed, verified, and proven correct.

The activity of proving formally that a system is correct is called *verification*. The verification of formally described protocols can, to some extend, be automated using interactive tools, such as theorem provers. However, the process is costly, since one has to write an input for a standard theorem prover from a protocol specification, and after proving it correct, must encode the protocol using a traditional programming language. Note that the last phase can be error-prone. Moreover, the verification information is not readily reusable as the system grows.

In effect, when new software must be developed quickly to meet ever growing market demands, doing formal verification is an exception rather then a usual practice. Therefore new methods are needed, which could be easier to deploy and less time costly.

2.1 Language technology for reliable systems

Although weaker than a full-blown program verification approach, the use of good programming abstractions and static analysis tools, such as type systems and model checking, can decrease the number of programming errors. It is therefore an effective way for increasing software reliability and reducing costs of the product development cycle.

In this paper, we focus on type systems and give some examples of properties that can be verified by typecheckers. Advanced type systems can be introduced either by building on top of existing programming languages, or by proposing new languages. Both approaches seem to be justified, as follows.

There is a great deal of old legacy software, which has been written using old traditional languages. This software must be still preserved, and adapted to meet new requirements. The legacy software is usually not documented. Software engineers spend therefore a great deal of time on trying to change and maintain it. Developing verification tools and methods could improve this process.

Recent advances in program static analysis have enabled to build tools that can catch a large number of programming errors in programs expressed using popular languages. For instance, the Extended Static Checker for Java (ESC/Java) [7] can detect, at compile time, common programming errors that ordinarily are not detected until run time, and sometimes not even then (e.g. null dereference errors, array bounds errors, type cast errors, and race conditions).

On the other hand, the advent of Internet computing, together with new emerging applications has spawned a lot of interests in developing new domain-specific programming languages. They offer programming abstractions that forbid certain programming errors. We sketch some example work below.

2.2 Domain-specific programming languages

The majority of existing systems have been built using traditional programming languages. Such languages do not provide good modularization facilities, no automatic memory management, and no good exception handling. This leads to code that is not easily penetrable, nor easy for modification, and almost impossible for reuse. Better programming languages can increase the productivity of programmers and reliability of new software. For example, ML [18] and Java have good modularization facilities, memory management, and exception handling.

New application domains may facilitate the process of old technology to be displaced. For instance, Java has become popular with the emergence of the World-Wide Web and applets, and has managed to displace C++ in many areas. The Ericsson example of the Erlang language [2] shows that efficient languages with good support for concurrent and distributed programming are being chosen instead of C, in the development of some large applications in the telecommunication industry.

Emerging application domains such as specification of security policies, web-oriented processing of semi-structured data, and mobile software have resulted in several new programming languages. For example, the Scala language [27] is aimed at web-oriented computing; the language provides support for processing of XML data and interoperability with Java at the bytecode level. Experimental programming languages such as the Join-calculus [11, 14] (implemented as an extension of OCaml [23]), TyCO [35], and Nomadic Pict [31, 43] have been implemented for experimenting with mobile computation.

Distributed programming with Nomadic Pict is an example of how the right choice of abstractions can help to shape the ideas and produce reliable code quickly. The Nomadic Pict constructs can be classified into two levels: location dependent primitives that require a programmer to know the current site of a mobile agent in order to communicate with it, and (at a high level) location independent primitives that allow communication with a mobile agent irrespective of any migrations. The two levels of abstraction provide clean separation between infrastructure (or system) programming and application programming. The former level is necessary, e.g. to build reliable messaging between agents [39], while the latter is concerned with the logical communication between mobile agents. This *separation of concerns* makes programming of applications much easier and less error-prone than if these two concerns would have to be dealt with at the same time.

While the technical arguments for new better programming languages are convincing, they are however still not sufficient for most industrial companies, and several non-technical hurdles must be addressed along the way. In this context, developing type systems seems particularly attractive, as the results of such work can often be applied to different languages. In particular, type systems that are developed in the context of strongly-typed languages such as Java and ML, may also contribute to the design of future industrial languages for building dependable systems.

3 What are the type systems?

The fundamental purpose of a *type system* [5] is to prevent the occurrence of runtime errors during the execution of a program. The definition of a runtime error is essential here. Traditionally, a type error arises when operations are applied to a wrong type of data. Type-safe programming languages, such as ML or Java, ensure that operations are only applied to appropriate values. However, the notion of a runtime error can be also understood in a much broader context. For instance, violation of some desirable properties, such as deadlock freedom, is also a runtime error, that can be prevented by devising a specialized type system.

A language is *typed* by virtue of the existence of a type system for it, whether or not type annotations actually appear in the program syntax. No mainstream language is purely implicitly typed, but in many languages, such as ML and Haskell [12], type information can often be omitted in large program fragments. However, keeping a moderate amount of type annotations is actually good. It can help to explain how or what code does. Since types are an integral part of the language syntax, programmers are forced to specify them, which seems a better way of documenting computer programs then adding *ad-hoc* comments.

3.1 The type-theoretic issues

Type systems are usually presented using a formal, mathematical definition. The formal definition of a type system comprises *judgments*, which are formal assertions about the typing of programs, *type rules*, which are implications between judgments, and *derivations*, which are deductions based on type rules. Such formalization of a type system is used for detailed proofs that a given programming language is *type sound*, i.e. that the absence of a well defined class of execution errors holds for all program runs that can be expressed within the language.

The basic motivation of a formal type system is however pragmatic. It provides precise, free from ambiguities, specification of a *typechecker*, which is usually part of a language compiler; it checks if there are no type errors. The implementation of a typechecker subsumes the use of specialized *type inference* algorithms that can find a type for an arbitrary program term within a given type system, if any type exists.

3.2 Advanced type systems

What interesting and desirable properties can be expressed as types and efficiently verified by a typechecker? Examples of recent work on type-based verification include: detection of race conditions and deadlock in concurrent programs, detection of communication deadlock, memory access control, fine-grain versioning to support interoperation of old and new code, type-based reasoning principles for mobile agents, type-safe marshalling of code, and type-safe separate compilation and linking. W describe some examples of the above results in the next section.

The advanced type systems, such as the above, use several novel techniques to make types more expressive (see [24, 25] for examples). We can mention here recent work on *dependent types*, which are constructed by extending types with expressions. Dependent types offer a powerful device for describing program's behaviour. For instance, extending an array type with the expression of value-range restrictions can be used by a typechecker for static elimination of runtime bounds checking on array accesses.

Another useful technique are *linear types*. The intuition behind linearity is that variables of linear type must be used exactly once. For instance, all the fields of a record, or tuple, must be used exactly once. We can use linear types e.g. to guarantee that fresh messages are delivered only once. A simple way of injecting typed values into the type level can be provided using *singleton types*, which are types of a single typed entity.

4 Typing for reliable distributed systems

There have been a lot of research on type systems in the programming language community. However, only recently there has been some successful work aimed at using types for verification of desired properties in concurrent and distributed programs. These properties appear to be especially important in the context of dependable software due to the error-proneness of this type of programming.

There is a growing interest in applying language-based approaches to problems that were traditionally regarded as system-level problems (e.g. deadlock detection) and solved dynamically at runtime. With the advances in the research on programming languages, particularly static analysis and compiler construction, they can be now solved statically at compile time. The "static" approach has not only the immediate advantage of detecting some of the programming errors, but also removes the computational cost of runtime checks and exception handlers since runtime errors (of a given kind) are guaranteed not to occur. Below are a few examples (not meant to be all-inclusive).

Types for correct communication on channels Recent work includes typing for *channel communication*. For example, Kobayashi *et al.* [15] give a linear type system for channel communication in the π-calculus; types of channels are marked with their direction (sending or receiving) and also with an information about how many times the channels can be used.

Session types developed by Vasconcelos *et al.* [38] allow us to specify high-level protocols as types. Session types define the types of data communicated by a distributed protocol, as well as message sequences of the protocol. The type-based protocol specification can be used to statically verify if the protocol's implementation is correct with respect to the specification.

Types for preventing data races Type systems are also used to statically detect common errors in *multithreaded programming*. For instance, *race conditions* can be avoided by using a type system of Flanagan and Abadi [10], which forces to protect each shared data structure with a lock. By imposing a partial order on lock types, their type system can also statically detect programs that could lead to *deadlock*.

Types for avoiding communication deadlock By adding usage expressions to types of communication channels, a process calculus of Sumii and Kobayashi [33] is able to detect a *communication deadlock*. The usage expressions indicate in which order communication channels can be used in a program; see also [15]. The type reconstruction algorithm in [16] allows the usage expressions to be omitted by annotating the sending and receiving operations with capabilities and obligations.

Types for safe program migration The Nomadic Pict project of Sewell, Unyapoth, and Wojciechowski [22] developed typed mobile process calculi with novel formal techniques: equivalences that take migration into account, and reasoning principles for agents that are temporarily immobile (e.g. waiting on a lock elsewhere in the system).

The typed calculi were used by Unyapoth and Sewell [36] for proving formally that example communication infrastructures for mobile computation (expressed in the Nomadic Pict language) are correct, i.e. all messages are correctly delivered to mobile agents despite of any agent migrations.

Types for safe distributed computation Another example of using advanced types in distributed programming is work by Sewell [29] on static typing support for modular wide-area programming, more precisely for separate compilation/linking. Interaction between programs may involve communication of values of abstract types; his type system provides the developer with fine-grain *versioning control* of these types to support interoperation of old and new code.

The Acute language [30] extends an OCaml core with support for type-safe interaction between separately-built programs. The main features are type-safe marshalling of values between distributed programs, and dynamic loading and controlled rebinding to local resources.

Types for dynamic software updating Hicks *et al.* [13] describe a calculus of type-safe dynamic updating of native code. More recently, Stoyle *et al.* [32] describe a core calculus for dynamic software updating in C-like languages; they define static updateability analysis that can infer program points at which all future (well-formed) code dynamic updates will be type-safe.

Types for safe code execution The work of Necula and Lee [20] has led to the idea of *proof-carrying code*, which allows the receiver of an untrusted program to verify quickly and reliably that the code is safe to execute, e.g. with respect to type and memory-access safety. They developed effective mechanisms for implementing *certifying compilers* [21], that can produce a machine-checkable proof of the code's safety.

Morrisett *et al.* [19] developed a typed assembly language capable of supporting many high-level programming constructs and also proof-carrying code. In this scheme, well-typed assembly programs cannot "go wrong".

5 Typing for safe network components

Types play an essential rôle in modular or component-based programming. Module systems for programming languages provide support for (among other things) name-space control, information hiding, and separate compila-

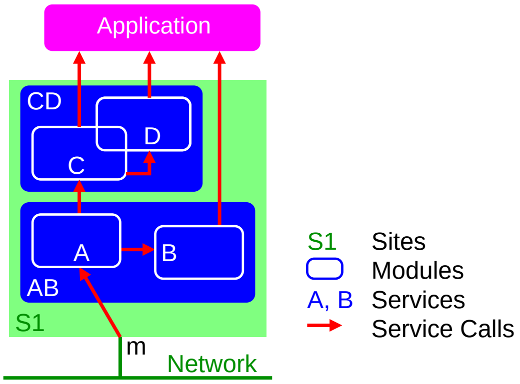

Figure 1. An example modular middleware.

tion. They allow to define fragments of code as modules (or classes in object-oriented languages) with interface definitions (also called signatures), and with the compiler automatically checking that object implementations actually conform to their signatures. Module systems, such as ML, provide us with flexibility in composing the code building blocks while preserving all the safety, error-checking, and compile-time consistency checking. Also Java provides safety and error- and consistency-checking, though not at compile time.

Types for composition correctness Let us consider *modular middleware*, as dependability issues should be clearly visible here. Different applications may demand different properties and guarantees related to e.g. fault tolerance and the quality of service. These varying properties can be provided by different middleware *services* that are composed from *protocol modules* (or *protocols*); see Figure 1.

However, not all compositions of individual protocols may have sense and those that do not, should be rejected. For dependability, we need to have *guarantees* that the protocol composition is correct. By *composition correctness*, we mean that the execution of services composed from components which are individually correct, matches the intended behaviour of these services.

Below we sketch a few example projects that aim at developing reliable networked applications from components with some guarantees on composition correctness. We also describe the limitations of each solution, and sketch how the design of more advanced type systems could help to overcome these limitations.

5.1 The FoxNet and Ensemble projects

The FoxNet project [4] pioneered the use of a module system to protocol design, demonstrating the use of a Standard ML for a modular implementation of TCP/IP. A protocol stack in FoxNet is built by composing modular protocol elements; each element independently implements one of the standard protocols. One specific combination of these elements implements the TCP/IP stack. However, other combinations are also possible and can be used to easily and conveniently build custom, non-standard network systems.

A limiting factor of this approach is that a standard module system does not take into account the semantics of protocols, which means that some compositions may not be correct. Ideally, in addition to checking structural properties of composition, some semantic constraints that one protocol may have placed on other protocols should also be verified.

The Ensemble project [37, 9] has addressed this issue in the context of *modular group communication* protocol stacks. A set of protocols is however fixed and so semantic composition can be easily checked at system start-up.

Types for semantic composition Another line of research are type-based approaches to semantic composition of *arbitrary* protocols. The idea is to extend declarations of modules with "contract types" that are sufficient to statically verify if a protocol composed from correct protocol modules conforms to the specification of the composite protocol. The "contract types" could build e.g. on Vasconcelos's session types and Kobayashi's communication types, which we have described in Section 4.

An example of advanced typing for semantic composition are also, in some sense, concurrency combinators (described in Section 5.3); they can be used to verify the order of service calls and internal parallelism.

5.2 The SAMOA protocol framework

The goal of the Crystall project [8] is to develop a novel group communication middleware [17]. To support protocol reuse, the middleware implementation uses *protocol composition* frameworks; they allow networked applications to be built from components that communicate using the framework's API. We have experimented with two protocol frameworks: Appia [1] and Cactus [6]. They however either do not support concurrency within a protocol (Appia), or depend on the facilities of the host language. Unfortunately, the latter means that the developer of a protocol module must be aware of the concurrency and synchronization context in which the module may be "plugged in". This makes programming tedious, and is a counter-example to the claim that protocol modules can be reusable.

We have therefore proposed *SAMOA* [28, 42] – a new protocol composition framework, with event-based communication, automatic concurrency control and message flow control. SAMOA has two features to support code reuse in the presence of concurrency. Firstly, event communication in our framework is *type-safe*, i.e. binding and rebinding of events to event handlers cannot result in runtime errors. To

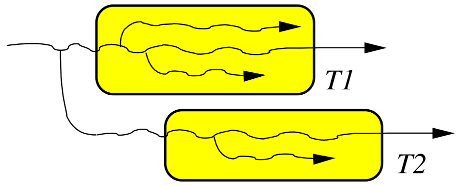

Figure 2. Concurrent, multithreaded tasks $T1$ and $T2$.

achieve this, we used *generics* – a new feature in Java. Secondly, it provides isolation-only transactions, called *tasks*, that help to implement concurrent protocols.

For instance, a task can be spawned to process a message through all protocol layers consistently, i.e. in *isolation* to any other concurrent tasks executed in the same stack. This property is analogous to isolation in database transactions. Tasks may be multithreaded, as shown in Figure 2.

Contrary to traditional transactions, tasks can freely perform operations that may have irrevocable input/output (I/O) effects, e.g. sending and receiving network messages. The main idea is to avoid the need for rollback at runtime (due to e.g., conflicts on task operations), by tightly controlling the order of such operations and guaranteeing that once started a task cannot run into conflicts.

Isolated tasks and type-safety of event (re)binding were useful to design support for *dynamic protocol update* [26].

Types for safe tasks Unfortunately, our experimental implementation of isolated tasks depends on data declared by the programmer, which compromises safety. To guarantee safety, we have therefore designed a type-safe language for isolated tasks [41]. The type system is used to verify data required by the basic versioning (pessimistic) concurrency control algorithm, which has been introduced in [42].

5.3 Declarative synchronization

Declarative synchronization promises to decrease the potential for programming errors, since the (error-prone) synchronization code is no longer entangled in the main program. We have designed two approaches to declarative synchronization: *static* [40] – in which the synchronization policy is defined in the form of a type expression that abstracts away from details of the synchronization code, and *dynamic* [34] – in which synchronization policies are defined as constraints between application-dependent *semantic rôles*, such as producers and consumers; the constraints are expressed using OCaml abstract types.

The former (static) approach allows us to express a synchronization policy using *concurrency combinators*, which are higher-order functions, or operators, that specify true parallelism (||), sequentiality (;), and isolation (isol and foll). Basic operators take as arguments symbolic names that can be bound (or assigned) to arbitrary code fragments. By combining operators, complex policies can be declared.

Concurrency combinators by example Below is a small example to illustrate the use of concurrency combinators. For clarity, we use a simple language, rather than a full-size language with ML-like modules or Java-like interfaces and classes. However, the approach can be applicable to these languages, too.

The following program first creates a reference cell object r initialized to zero (the operation !r is used to read the content of the object r, and the operation := to overwrite it). After creating the object, the program defines two functions: a function a, that overwrites object r with its argument, and a function b that simply reads the current content of r. The implementation of each function is bound (using the # construct) to component or *service* names A and B. (We can think of service names as corresponding to module or interface signatures; they will be used to define combinators.)

```
let r = ref 0 in
let a x = A # (r := x; ()) in   (* update the store *)
let b () = B # !r in  (* read the store *)
fork (a 1); (b)   (* this may return 0 or 1 *)
```

The main body of the program calls the two functions in parallel and returns the value returned by function b. The function a gets 1 as its argument and is called by a separate thread that is spawned using the construct fork. Since threads can be arbitrarily interleaved, the program's execution returns either zero or 1.

We can use the concurrency combinators to *declare* desirable program behaviour without actually modifying the code of our components. Below is a program that allows the functions to be called by separate threads but the outcome of this execution is like service B would be called (and returned) before A. This behaviour is expressed using a combinator B foll A, which means *execute concurrent A and B in isolation, with the (ideal) serial run "A followed by B"*.

```
B foll A  (* read old content, then update *)
let r = ref 0 in
let a x = A # (r := x; ()) in   (* update the store *)
let b () = B # !r in  (* read the store *)
fork (a 1); (b)   (* this returns 0 in every run *)
```

However, this means that the program will always return the *old* content of the object, since function b that is bound to a service name B and used to return the object's value, is always called before calling function a (that is bound to A).

Typing for composition safety Let us extend our program by adding another function c (bound to a service name C), which will be used to "filter" the argument passed to function a (implementing service A). For simplicity, we omitted function types; we assume that function c has a polymorphic type, i.e. it can accept *any* argument.

```
C; (B foll A)  (* filter input, read old content, then update *)
let r = ref 0 in
let a x = A # (r := x; ()) in
let b () = B # !r in
let c x = C # x in   (* filter the argument *)
fork (a (c 1)); (b)   (* this returns 0 in every run *)
```

Note that we have also extended the concurrency combinator declaration to C; (B foll A), so that the program is correct only if the filter C is called before services A and B are called; this behaviour is expressed using ; (semicolon).

An important feature of the concurrency combinators language is *composition safety*. We have developed a type system [40] that can *guarantee* that a synchronization policy declared using combinators can be actually satisfied by program execution.

For example, the following two versions of our example program do not typecheck.

```
C; (B foll A)  (* filter input, read old content, then update *)
let r = ref 0 in
let a x = A # (r := x; ()) in
let b () = B # !r in
let c x = C # x in
(* fork (a (c 1)); (b)   this returns 0 in every run *)
fork (c (a 1)); (b)
```

The above program is not correct. The problem is that we have declared service C to be called *before* calling services B and A. However, the program implementation (see the highlighted text) does not match this specification since the update service A is called *before* the update argument is filtered using service C, thus violating the desired property expressed using the concurrency combinator.

Now let us go back to the previous version of the program, but we replace the concurrency combinator by a new combinator (B foll A) and A || C. The new combinator checks if services A and C can be executed in parallel (e.g. in order to support multi-processor architectures):

```
B foll A
A || C
let r = ref 0 in
let a x = A # (r := x; ()) in
let b () = B # !r in
let c x = C # x in
fork (a (c 1)); (b)
```

The program however does not type check either. We requested services A and C to be executed in parallel by separate threads (see the highlighted text). However, this requirement cannot be satisfied by *any* runtime execution of the program, since the filter service C and the update service A have been implemented so that they are called sequentially by the same thread.

Thus, concurrency combinators can be used to verify if certain conditions hold in programs without actually executing the programs.

6 Conclusion and future work

Software testing is often useful to find common errors, but is not capable of guaranteeing correctness. On the other hand, a full-blown program verification approach using theorem provers is time costly and the verification information is not readily reusable as the system grows.

We argue that type-based verification, that falls in between the two, provides a useful method for satisfying dependability with respect to given properties. A typechecker can guarantee that certain desirable properties, expressed as types, are always satisfied by program execution.

In the paper, we gave examples of properties that can be type checked in programs. It is not clear yet what other properties of dependable systems could be verified in this way. We hope that future research would soon contribute to answering this question. For instance, it is not obvious how much of program's behaviour can be easily expressed using types, and efficiently verified by a typechecker.

Many program properties are not static but *ephemeral*, i.e. they can dynamically change during program execution. In our future research, we would like to investigate combining static methods with *dynamic checking*, which may turn out to be a reasonable solution for guaranteeing safe execution of distributed programs. In this approach, the compiler would emit the (correct) code for some run-time checks.

7 Acknowledgments

This work was supported by Swiss NSF contract #21-67715.02 and Hasler Stiftung project DICS-1825.

References

[1] *The Appia project*. http://appia.di.fc.ul.pt/.

[2] J. L. Armstrong and S. R. Virding. Erlang – An experimental telephony switching language. In *Proceedings of the XIII International Switching Symposium*, May 27–June 1, 1990.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition*. Addison Wesley, 2000.

[4] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in standard ML. *Higher Order and Symbolic Computation*, 14(4):309–356, 2001.

[5] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[6] *The Cactus project*. http://www.cs.arizona.edu/cactus/.

[7] Compaq Research. *The Extended Static Checking for Java (ESC/Java)*. http://www.research.compaq.com/SRC/esc/.

[8] *The Crystall project*. http://lsrwww.epfl.ch/crystall.

[9] *The Ensemble toolkit*. http://dsl.cs.technion.ac.il/projects/Ensemble/.

[10] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of ESOP '99: the 8th European Symposium on Programming*, volume 1576 of *LNCS*. Springer, Mar. 1999.

[11] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96: the 7th International Conference on Concurrency Theory*, volume 1119 of *LNCS*, Aug. 1996.

[12] *The Haskell language*. http://www.haskell.org/.

[13] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proceedings of PLDI '01: the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[14] *The JoCaml language*. http://pauillac.inria.fr/jocaml/.

[15] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5), 1999.

[16] N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR 2000: the 11th Conference on Concurrency Theory*, volume 1877 of *LNCS*, 2000.

[17] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of ACM/IFIP/USENIX Middleware '03*, volume 2672 of *LNCS*. Springer, June 2003.

[18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[19] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[20] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI '96: the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[21] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of PLDI '98: the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[22] *The Nomadic Pict project*. http://www.cl.cam.ac.uk/users/pes20/nomadicpict.html.

[23] *The Objective Caml language*. http://caml.inria.fr.

[24] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[25] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, MA, 2005.

[26] O. Rütti, P. T. Wojciechowski, and A. Schiper. Dynamic update of distributed agreement protocols. Technical Report IC-2005-012, Ecole Polytechnique Fédérale de Lausanne (EPFL), Mar. 2005.

[27] *The Scala language*. http://scala.epfl.ch/.

[28] *The SAMOA toolkit*. http://lsrwww.epfl.ch/samoa.

[29] P. Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of POPL '01: the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[30] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, Oct. 2004. Also published as INRIA RR-5329.

[31] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages*, volume 1686 of *LNCS*. Springer, Oct. 1999.

[32] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *Proceedings of POPL '05: the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2005.

[33] E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proceedings of HLCL '98: the 3rd International Workshop on High-Level Concurrent Languages*, volume 16.3 of *ENTCS*, 1998.

[34] V. Tanasescu and P. T. Wojciechowski. Role-based declarative synchronization for reconfigurable systems. In *Proceedings of PADL '05: the 7th Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *LNCS*, Jan. 2005.

[35] *The TyCO language*. http://www.ncc.up.pt/tyco/.

[36] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proc. of POPL '01: the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2001.

[37] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. Technical Report TR97-1638, Cornell University, Computer Science, July 1997.

[38] V. T. Vasconcelos, A. Ravara, and S. Gay. Session types for functional multithreading. In *Proceedings of CONCUR '04: the 15th International Conference on Concurrency Theory*, volume 3170 of *LNCS*. Springer, 2004.

[39] P. T. Wojciechowski. Algorithms for location-independent communication between mobile agents. In *Proceedings of AISB '01 Symposium on Software Mobility and Adaptive Behaviour*, March 2001.

[40] P. T. Wojciechowski. Concurrency combinators for declarative synchronization. In *Proceedings of APLAS '04: the 2nd Asian Symposium on Programming Languages and Systems*, volume 3302 of *LNCS*, Nov. 2004.

[41] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proceedings of PPDP '05: the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.

[42] P. T. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2004.

[43] P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2), April–June 2000.