PEPITO

IST-2001-33234

PEer-toPeer Implementation and TheOry

Workpackage 1: Formal Models

Deliverable 1.7
First Progress Report on Formal Models

(Covering period 2002.01.01 – 2002.12.31)

January 3, 2003

Prepared by Peter Sewell, with input from Vincent Cremet, Uwe Nestmann, James Leifer, Andrei Serjantov, Peter Van Roy, and Keith Wansbrough. Sites INRIA, EPFL, UCAM, and UCL were involved in this work.

The Technical Annex contains three tasks in this workpackage:

1. Transactions and the Semantics of Failure [EPFL 18, UCAM:18, UCL:12]
2. Versioning and Modularity [UCAM:24]
3. Models of Peer-to-Peer Group Collaboration [EPFL:6, INRIA:6, UCAM:10, UCL:6]

We discuss each in turn, concluding by listing the associated papers published during this year.

1 Transactions and the Semantics of Failure

Overview Work in this task has proceeded at three levels of abstraction. (1) At the lowest level we are building a precise semantic model of failures as they occur in distributed communication, specifying the behaviour of the UDP and TCP protocols, with their *sockets* library interface. A description of our UDP specification was published this year [WNSS02, NSW02]. As for TCP, we have completed a first-draft specification and built a suite of validation tools; this work is ongoing. (2) These low-level protocols do not provide good *failure detection* to applications. We have worked on process-calculus models of Chandra-Toueg style failure detectors and a Consensus algorithm using them, allowing one to carry out proofs about the algorithm that are formally justified by the process-calculus semantics. (Related work on a model implementation of such failure detectors, above our specified UDP layer, is described in WP2). (3) At the highest level, applications may need *transactional* idioms for programming in the presence of failure. Work has proceeded on three lines: on an axiomatisation of desired transactional properties, on a model OCaml implementation of distributed transactions, and on describing the behaviour of the UCL implementation.

1.1 TCP Failure Semantics

(Peter Sewell, Keith Wansbrough, Steven Bishop, Michael Norrish; UCAM) The protocols TCP and UDP are ubiquitous, forming the basis of almost all communication on the Internet. Over the years since their inception, they have been tweaked and improved in many ways by many parties. The resulting system evidently works, but it is poorly understood by the vast majority of those who use it. While recipes for common uses abound, there are very few sources of information on the exact behaviour of the system in specific cases. In particular, the precise response of the system to failure, in the form of erroneous parameters, attack, media or system failure, etc., must be determined by experiment or by inspection of an implementation's source code. This is particularly important for P2P systems, which must be robust but depart from the common client/server usage of TCP. The documentation that is available is arbitrarily organised, voluminous, and entirely in natural language, with all the ambiguity and incompleteness that entails. Furthermore, implementations often differ in subtle ways, usually undocumented and discoverable only by experiment.

We are therefore developing a formal model of these protocols, in order to precisely understand their behaviour. This model provides detailed answers to questions such as "how long a network interruption will TCP tolerate, and what effect will there be on throughput after reconnection?" or "what error codes must an application be prepared to deal with when making a connect system call?". The model is based upon the available documentation in conjunction with the source code of the BSD reference implementation. Crucially, however, it is being *experimentally validated* against the most common implementations; a key part of our work is the experimental discovery of differences in behaviour between these implementations.

Our formal semantics of UDP has been published this year [WNSS02, NSW02] (the associated software library has been used by Leifer, at INRIA, in an implementation of failure detectors). We have now moved onto the rather larger task of TCP. The first draft of the TCP model is complete (12000 lines, or 100 typeset pages). Much work has gone into untangling the historical accumulation of modification and extension, yielding a semantics that is (relatively) intelligible and clear as well as precise.

Using the HOL theorem prover has allowed us to verify type-correctness of the model (internal consistency), and will also allow us to validate the model against experimental results as well as verifying various sanity properties. The large size of our model has necessitated our making various improvements to the HOL system. In addition, we have extended HOL with a symbolic evaluator which we apply to our model in order to validate it against experimental results.

Experimental validation involves setting up a test network, building a test harness, generating a set of tests, performing them, and validating the model against the test results. We have set up a test network comprising a number of machines running each of the common implementations (Linux, FreeBSD, Windows XP). The test harness allows remote execution of system calls, remote white-box inspection of protocol state where possible, network packet injection, and network packet sniffing, all under the control of a test script; it also logs all events and generates a trace suitable for input to HOL. This significant development effort is essentially complete. In early 2003 we will begin generating tests, at first manually but then semi-automatically, and will use the results of validation to fine-tune our model and to parameterise it by implementation. We also hope to use the symbolic evaluator to link the model's inputs and outputs to the packet sniffer and injector, thus emulating another machine.

Once our formal model is complete and validated, we intend to prepare an annotated version for publication and dissemination. We expect this document to be of use for implementation of distributed systems, especially those that must be failure-aware, reliable, and multi-platform ones. It should also be a reference for those building new implementations of, or extensions to, TCP.

1.2 Failure Detectors and Consensus

(Uwe Nestmann, Massimo Merro, Rachele Fuzzati; EPFL) We continued to work on the process-calculus model of Consensus using Chandra-Toueg style failure detectors. The main result is an intermediate representation for describing the reachable states of a consensus system in a global round-oriented manner: at any moment, i.e., in any reachable state the intermediate representation makes precise the history of the current run in terms of all the messages that have so far been sent and, most importantly, organized according to the various rounds that the algorithm has passed. It is this kind of global view of a system that is acting as a vehicle for most proofs about distributed algorithms, while it was never formalized, but rather remained vague. Based on the process calculus, we establish a formal correspondence between the local (process-oriented) and global (message- and round-oriented) views of the system such that the former acts as a formal validation for the latter. This allows us to carry out the proofs along the lines of Chandra Toueg, but now formally justified by the operational semantics of our process calculus. There should be a Technical Report available by the time of the review.

Apart from this application-oriented work, we have also modeled the other failure detectors of Chandra and Toueg. We are currently working on the formal comparison of our representation to theirs. This work is actually independent of any calculus or language describing the algorithms that make use of failure detectors.

For the next year, we plan to study extensions of our operational semantics setting for failure detectors towards more dynamic systems as they appear in p2p-environments.

1.3 Transactions - Axiomatisation

(Vincent Cremet, Andrew Black, Martin Odersky, Rachid Guerraoui; EPFL) With the aim of giving a formal basis to transactions, we have decomposed the properties that are usually required together in the definition of a transaction, namely atomicity, isolation and durability, and reified them in the syntax of a calculus. So, in addition to the classical constructs for parallel composition, sequential composition and non-deterministic choice, the calculus contains a special construct for each transactional property, as well as a construct that expresses a crash/recovery of the system.

In this setting, we are able to state and prove that the execution of a program, composed of transactions which individually do not violate the consistency of the system, will preserve this consistency – even in the presence of crash/recoveries. Here we consider that a term which consists simply of a sequence of consistent actions, i.e. without parallelism or occurrences of crash/recovery events, preserves the consistency of the system. We give semantics to our calculus by means of an order relation between terms, which roughly represents trace inclusion. We then prove that each term has an ancestor (with more traces) which has the previous "consistent" form, which concludes the proof.

We are now trying to give a practical interpretation to this work, to see how it could be useful to reason about existing transactional systems, and if we could include in a real language the ACID primitives of our calculus. We would like also to add to the language the notion of communication, which is required for cooperation between transactions, as well as the notion of replication. Of course, ultimately we will also have to deal with distribution. A draft paper is available [CBOG02].

1.4 Transactions – Model OCaml Implementation

(Peter Sewell; UCAM) We have built (in OCaml) a model implementation of distributed transactions, in order to clarify exactly what the essential components and required language primitives are. The hope is that this will provide a starting point for a semantically-cleaner development of transactional programming in the future.

1.5 Transactions – UCL

We are working on both models for transactions in a P2P framework. We intend eventually for the models to mirror faithfully what our implementations are doing (and to show the way to improving the implementations), but we are far from achieving this goal at the current time.

2 Versioning and Modularity

Overview When different components of a distributed system can interact, by communicating values, the sender needs to *marshal* data, i.e. transfrom the data into a bit sequence and the receiver needs to *unmarshal* the data, i.e. reconstruct the original value from the serialised form. Most programming languages have some support for these two operations, but several problems have not been well-understood.

Firstly, the values may refer to local resources that must be *dynamically rebound* to resources at the receiver. Secondly, the sender and receiver – which may be of different versions – must have 'sufficiently consistent' views of the *type* of the communicated values, to avoid later runtime errors. This is particularly problematic in languages with abstract types. The necessary coexistence of multiple versions in a large-scale distributed system comes from the fact that simultaneous installation of new software is often impractical: one must be able to dynamically update the components on some nodes, whilst retaining interoperability. A third problem arises for systems that require *dynamic updating* of individual modules within a single node. Such modules can interact not only by communication, but also by function or method calls (with sufficiently expressive communication, the other problems would be essentially this case).

In this task we have studied each of these three problems, developing model programming languages (with type systems and operational semantics) that indicate how the problems could be addressed in a full language design. In each case there is a tension between expressiveness and control – one would like to detect any errors as early as possible, ideally at compile-time, but sometimes that is impractical.

The technical annexe divided this work into the UCAM *Versioning and Modularity* task of WP1 and the INRIA *Resource Control* task of WP3. In the event that division has turned out to be rather artificial – there is an interlinked collection of problems to be addressed, and we have had considerable discussion of them between the sites. In this Periodic Progress Report we describe various work in lambda-calculus settings here, and the work of Alan Schmitt, in a join-calculus setting, in WP3.

2.1 Dynamic Rebinding

(Gavin Bierman, Peter Sewell, Gareth Stoyle, Keith Wansbrough; UCAM, and Michael Hicks; University of Maryland) Most programming languages adopt static binding, but for distributed programming an exclusive reliance on static binding is too restrictive: dynamic binding is required in various guises, eg when a marshalled value is received from the network, containing identifiers that must be rebound to local resources. Typically it is provided only by ad-hoc mechanisms that lack clean semantics.

In this work we have adopted a foundational approach, developing core dynamic rebinding mechanisms as extensions to simply-typed call-by-value (CBV) lambda-calculus. To do so we first explored refinements of the CBV reduction strategy that delay instantiation, to ensure computations make use of the most recent versions of rebound definitions. We introduced *redex-time* and *destruct-time* strategies. The latter forms the basis for a *lambda-mark* calculus that supports dynamic rebinding of marshalled values, while remaining as far as possible statically-typed. We extended lambda-mark with concurrency and communication, giving examples showing how wrappers for encapsulating untrusted code can be expressed. Finally, we showed that a high-level semantics for dynamic updating can also be based on the destruct-time strategy, defining a *lambda-update* calculus with simple primitives to provide type-safe, dynamic updating of code. We have a direct implementation of the destruct-time lambda semantics, and are working on realistic compilation strategies. We thereby established primitives and a common semantic foundation for a variety of real-world dynamic rebinding requirements.

A technical report on this work should be completed early in 2003.

2.2 Marshalling and Abstract Type Versions

(James Leifer, Gilles Peskine, Jean-Jacques Lévy; INRIA. Peter Sewell, Keith Wansbrough; UCAM) In this work we again are looking at the design of type-safe language constructs for manipulating marshalled data in a distributed setting. Here, though, our focus is on the appropriate notion of *type equality* for use between components of a distributed system. In particular, we have considered how to encode sufficient type information in the serialised form for the unmarshal primitive to verify at run-time the compatibility between the actual type of the sent value and its expected type. This is complicated by the rich nature of distributed communication in ML-like languages (Ocaml and Jocaml). A key goal is to detect errors as early as possible (particularly important in hard-to-debug distributed systems). Ideally, one would like to detect errors at compile-time. Failing that, we aim to detect errors at the latest at unmarshal-time, rather than depend on general dynamic typing.

One of the most powerful features of ML-like languages are *modules* for packaging data types and code. A module may hide the representation of the exported data types, thus making them *abstract*. By doing this, the programmer increases code maintainability since the type system enforces the requirement that no code outside the module may depend on the concrete representation of the abstract types. Likewise, the programmer may rely on invariant properties of data of an abstract type (for example, representations of sets as arrays whose elements are always sorted in ascending order) knowing that no other functions outside the module can manipulate these values.

Thus preserving the abstraction barrier presented by a module's interface is a key aspect for any ML-like language and is exactly the property broken by a naive marshaling primitive that stores with each value its concrete representation type: in such a situation, a receiver may expect a marshaled value to have an abstract type, and thus satisfy the invariant properties maintained by the module that defines that type. Yet the sender may have constructed the value in such a way that breaks all invariants assumed by the reader.

We have created a calculus and type system to model the subtle properties of abstraction preservation for ML-like languages with marshaling. Our calculus is based on a lambda-calculus enriched with a module system (related to Harper and Lillibridge's sytems with singleton kinds), and with primitives for marshal, unmarshal, and communication. The type system for the calculus makes every abstract type distinct (as is common for generative type systems in ML) but introduces a careful "escape" via equality properties of module *hashes*. The reduction rules transform abstract types into hashes that record a fingerprint of the module (and thus of the abstraction preserving functions for manipulating it) – essentially, hashes provide a truly global namespace for abstract types. In this way, programs may read values of abstract types generated via a module on one machine and manipulate them via modules declared locally, checking only that the module hashes correspond at unmarshal-time, and ensure that abstraction is preserved.

We have proved a "progress" result that shows that the system never gets stuck and have completed most of a "soundness" result showing that well-typed programs reduce only to well-typed programs.

We have also considered calculi for modelling more subtle variations of abstraction preservation that explicitly allow the *version* of the module at the sender side to differ from the version at the receiver side. Work on this is ongoing.

Gilles Peskine and James Leifer (INRIA) have also implemented a prototype modification of the current Ocaml compiler that adds a type-safe marshall and unmarshall operation.

The marshal primitive is augmented to output a run-time type representation and the unmarshal primitive performs a dynamic run-time check. This work does not integrate the theoretical results on module hashing above, though that is the eventual goal. The implementation represents an important first step since it involves a detailed analysis of the internals of Ocaml (a full-scale compiler/run-time system) and careful treatment of Ocaml's rich types (records, objects, variants).

2.3 Run-time Version Update

(Gavin Bierman, Peter Sewell, Gareth Stoyle; UCAM. With Michael Hicks; University of Maryland) In this ongoing work we are exploring Dynamic Software Updating (DSU) in more detail, building on the previous two items.

A number of DSU systems have been designed, each in some fashion enabling running programs to be updated with new versions of code and data without interrupting their execution. There is, however, still little rigorous understanding of how to use DSU technology so that updates are safe.

As first steps in this direction, we have defined two small *update calculi* with precise mathematical semantics. They are based on the lambda-calculi of **Dynamic Rebinding** above. The first supports updating technology similar to that of the programming language Erlang. Here new versions of a module can be *added* to a running system, and there is fine-grain control for how execution changes over from old to new code. A paper describing this has been submitted for publication, with details of the calculus, examples of its expressive power, and discussion of how it might be used or extended to guarantee safety properties.

Our second update calculus supports updating of the implementations of abstract data types, reminiscent of the language Dynamic ML. For this, it has been necessary to use (a modified form of) the semantics we developed in **Marshalling and Abstract Type Versions** above, preserving abstraction as computation proceeds.

3 Models of Peer-to-Peer Group Collaboration

Overview For this task, rather than follow an *ab initio* approach of defining a single new calculus, our approach during the year has been more bottom-up – in part for scientific reasons, as this has seemed a more fruitful direction, and in part as a consequence of recruitment difficulties. The results outlined in previous tasks, on failure semantics, consensus, transactions, and dynamic binding, have addressed particular aspects of the problem. We also have work focussed not on arbitrary P2P systems but on P2P systems providing particular *anonymity* properties, which we outline below, and some preliminary work on models for P2P collaboration (UCL). Here the intention is that the models faithfully mirror what our implementations do, and show the way to improving the implementations, but this is far from complete at the present time. Some initial discussions on

P2P calculi were had between Uwe Nestmann (EPFL) and Peter Sewell (UCAM).

3.1 Anonymity

(Andrei Serjantov, Peter Sewell; UCAM, with collaborators elsewhere) Andrei Serjantov's PhD work is focussed on the analysis and design of systems for anonymity and censorship resistance, many of which use peer-to-peer architectures – in the form of distributed hash tables and/or of networks of *mixes* which can exchange messages.

A censorship resistant system is a distributed system for making documents available, whilst ensuring that they are hard to remove or make inaccessible. Serjantov designed a censorship resistant system based on a P2P distributed hash table architecture, which provided protection to the storers of documents as well as publishers and retrievers. This work was presented at the International Workshop for Peer to Peer Systems in March 2002 [Ser02].

Turning to systems for anonymous communication, it is particularly challenging to understand what level of privacy is guaranteed by a system, under various assumptions on the power of an attacker. Rigorous analysis is therefore crucial. Together with George Danezis, Serjantov developed a new measure of anonymity based on probability distributions rather than the more traditional anonymity sets and showed how it applied to single mixes and mix systems – areas where the more traditional approaches fail. This work "Towards an Information Theoretic Metric of Anonymity" was presented at the Privacy Enhancing Technologies Workshop (PET) in April 2002 [SD02]. Following on from this, Roger Dingledine, Andrei Serjantov, and Paul Syverson looked at various kinds of mixes which have been developed over the past few years and analysed their property in terms of the new metric. In particular, this work focusses on how resistant they are to blending or (n-1) attacks. This was presented at the Information Hiding Workshop in October 2002 [SDS02]. With Richard Newman, Serjantov has developed a technique to analyse timed pool mixes. Based on the probabilistic metric and simulation, the technique allows these mixes to be compared to threshold mixes. This has been accepted to the Working Group 11.4 meeting on Privacy and Anonymity Issues in Networked and Distributed Systems [SN03].

Other work on the analysis and design of mix systems is ongoing, and we have made contributions to the implementation of a new generation system.

4 Access Control

(Moritz Becker, Peter Sewell; UCAM) Unforeseen in the Technical Annex, we have also worked on the foundations of access control in distributed systems, in the PhD work of Moritz Becker. In a distributed system, and particularly in a P2P system, something better than traditional access control models is required. We are studying policy- and role-based access control models, in which a security policy is an explicit entity, expressed in a logic. There is a tension in the design of such a logic, however, between expressiveness and decidability. Focussing on constraint logic programming, we have proved decidability of a range of constraint domains, sufficient for expressing a wide class of policies, and have a prototype OCaml implementation working. There is interaction here with members of the SECURE project.

References

[CBOG02] Vincent Cremet, Andrew Black, Martin Odersky, and Rachid Guerraoui. Axiomatization of transactions. `http://lampwww.epfl.ch/~cremet/transactions.html`. Paper draft, 2002.

[NSW02] Michael Norrish, Peter Sewell, and Keith Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, September 2002. `http://www.cl.cam.ac.uk/users/pes20/Netsem/sigops-ew2002.ps`.

[SD02] Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In Paul Syverson and Roger Dingledine, editors, *Privacy Enhancing Technologies*, LNCS 2482, San Francisco, CA, April 2002. `http://www.cl.cam.ac.uk/~aas23/papers_aas/set.ps`.

[SDS02] Andrei Serjantov, Roger Dingledine, and Paul Syverson. From a trickle to a flood: Active attacks on several mix types. In Fabien Petitcolas, editor, *5th Workshop on Information Hiding*, LNCS 2578, October 2002. `http://www.cl.cam.ac.uk/~aas23/taxonomy.pdf`.

[Ser02] Andrei Serjantov. Anonymizing censorship resistant systems. In A. Rowstron P. Druschel, F. Kaashoek, editor, *International Workshop on Peer to Peer Systems*, LNCS 2429, Boston, MA, March 2002. `http://www.cl.cam.ac.uk/~aas23/papers_aas/Anon_p2p2.ps`.

[SN03] Andrei Serjantov and Richard Newman. On the anonymity of timed pool mixes. In *Working Group 11.4 – Privacy and Anonymity Issues in Networked and Distributed Systems*, Athens, Greece, May 2003. To appear.

[WNSS02] Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proceedings of ESOP 2002: European Symposium on Programming (Grenoble) LNCS 2305*, pages 278–294, April 2002. `http://www.cl.cam.ac.uk/users/pes20/Netsem/timing-udp-a4.ps`.