Migrating Protocols In Multi-Threaded Message-Passing Systems

Austin Anderson
University of Southampton
ama08r@ecs.soton.ac.uk

Julian Rathke
University of Southampton
jr2@ecs.soton.ac.uk

Abstract

Dynamic software update is a technique by which a running program can be updated with new code and data without interrupting its execution. Often we will want to preserve properties of programs across update boundaries. Preserving simple typing across update boundaries for single-threaded programs is well studied. There are other higher-level properties we may wish to preserve, particularly for multi-threaded programs. Session typing is used to guarantee that a set of parallel threads communicate according to a given protocol. Hence we investigate preserving correct communications behaviour of a set of parallel threads correctly across update boundaries which change the running protocol. We present a procedure for updating multiple threads to cleanly migrate a system from one protocol to another.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming - program modification, program transformation; D.1.3 [*Programming Techniques*]: Concurrent Programming - multi-threaded programming; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Typing

Keywords Dynamic Software Update, Multi-Party Session Types, Distributed-Protocols

1. Introduction

Dynamic software update is a technique by which a running program can be updated with new code and data without interrupting its execution [13]. Generally there are various behaviours which are considered safe or unsafe, in a given situation. We can use safety properties to guarantee that we do not perform any unsafe actions. Simple type safety is an obvious example of such a property and preserving simple type safety for updates to single threaded programs is well studied [8, 13]. In a multi-threaded system there are more complex behaviours and hence more complex safety properties which cannot be reasoned about using simple type safety alone. The inability to provide safety for multi-threaded systems is a significant restriction of the analyses for single-threaded systems.

HotSWUp'09, October 25, 2009, Orlando, Florida, USA.

Multi-threaded session typing is a type discipline used to ensure that a set of parallel threads communicate according to a specific protocol [3]. In our earlier work we present a mechanism to update individual threads in a multi-threaded system such that the updated thread will continue to communicate according to the correct protocol after the update as before the update [2]. We refer to this property as *update safety*. In that work we present a static analysis which we use to guarantee (and prove) subject reduction and fidelity and linearity safety properties. The most important safety property is that of fidelity, which states that for each possible communication action for the code we have a corresponding possible step in the protocol. In that work an updated thread continues with the same protocol and hence the update is invisible to the other threads. We refer to such an update as an internal update. Our contribution in that work is to permit internal updates to threads which can change the function bodies of named functions. We also extend the session typing discipline by permitting non-tail-recursive function calls.

Migrating a multi-threaded system from one protocol to another requires that we update multiple threads; we clearly cannot change the communications behaviour of a thread running one end of a protocol without changing the other end. We refer to such updates as *system updates*. We refer to the updates to individual threads as *thread updates*. A system update includes multiple thread updates. Hence we also discuss system updates in the context of multiple updates, as a system update will update multiple threads. If we use a methodology of applying the thread updates from a given system update separately, migrating each thread independently from the old protocol to the new, we can produce errors. We present an example error which can occur from such a methodology. Hence in order to cleanly migrate from the old protocol to the new we must provide some coordination of the applications of the updates to individual threads.

Our contribution in this paper is a coordination procedure which should rule out errors of the form we demonstrate. We rely on our static analysis which guarantees subject reduction and preservation of fidelity and linearity. We do not present this analysis; it is essentially the analysis we presented in our prior work [2]. Using our coordination procedure we should be able to safely migrate a multi-threaded system from one communications protocol to another.

The rest of the paper is structured as follows. In Section 2 we describe our language and global session types which we use to describe protocols. In Section 3 we present two examples to motivate the problem and demonstrate the type of error we wish to rule out. In Section 4 we describe our proposed solution, which includes a distributed protocol to determine when each thread should apply an update. In Section 5 we present related work. In Section 6 we conclude and present future work.

2. Language and Types

Our language is a first order, call-by-value, functional programming language with named functions. We provide communications primitives which request and accept starting a session, send and receive values, and select and accept selection of services (a path for the protocol to take out of several options). These primitives are based on those presented in the session typing literature [3]. We also provide let expressions and if expressions. Sequential composition $e_1; e_2$ is considered syntactic sugar for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ where x is free in e_2. Finally we also include the update command. The update which is applied is not specified in the code; if it were it would not be a dynamic update. The update command is an update point which applies, if it can, any update which is relevant to the thread in which it is executed. If there are no relevant updates then we simply perform skip. An update is provided at run time by an out-of-band communications mechanism.

In our system we do not communicate between threads, but between roles. A role can only be held by a single thread at any one time, but a role can be transferred from one thread to another, permitting us to describe some high level communications behaviours (higher order session typing). We provide the definitive description of a communications protocol that a group of roles follow using *global session types*. We define global session types, ranged over by G, in the folowing grammar:

G	$::=$	$p \to \underline{p} : \langle U \rangle$	Send a value of type U where $p \notin \underline{p}$
	$\mid$	$p \to \underline{p} : \{l_i\ :\ G_i\}$	Send a label in $\{l_i\}$ where $p \notin \underline{p}$
	$\mid$	$G.G$	Sequential composition
	$\mid$	$\mu \mathbf{t}.G$	Recursion
	$\mid$	$\mathbf{t}$	Recursion variable

The type $p \to \underline{p} : \langle U \rangle$ denotes that role p will send a value of type U to roles $\underline{p}$ (a set of roles which does not include p). The type $p \to \underline{p} : \{l_i\ :\ G_i\}$ denotes that role p will send a label l_i to roles $\underline{p}$. After sending label l_i the session will continue with session type G_i. This permits us to permit the protocol to proceed using one of several specified sub-protocols. The type $G_1.G_2$ denotes sequential composition; the global session will first perform G_1 then G_2. The type $\mu \mathbf{t}.G$ denotes recursion, on recursive variable $\mathbf{t}$. As protocols can include more than two roles multi-party communications can be represented by a single global session type. We refer to global session types and protocols interchangeably. The protocol could be inferred from the code, but we require the programmer to explicitly state the global session type for each session. Our global session types are based on those in the literature [3, 9]

We use a global queue for handling messages. When a thread is sending a value it will append the message on to the end of the queue. When a thread is receiving a value it will search through the queue from the front, removing the first message for the role on which the thread is receiving. This provides asynchronous communication where a receiving thread can block to wait for a message to be put on the queue, but sending threads do not block waiting for a receiving thread to be ready. Along with our session typing system presented in our proper work [2] this permits us to guarantee the order in which messages will be received and hence the types of the values being received. Informally this type system guarantees that for each send there is a complementary receive (i.e. for three sends which send, in order, an integer then a boolean then an integer then there will be three receives, expecting exactly those types in that order).

$$P_1 \| P_2 \qquad P_1 = \langle i, d_1 \rangle \qquad P_2 = \langle j, d_2 \rangle$$

$$d_1 = \mathsf{fun}\ f(x) = e_1\ \mathsf{in}\ \mathsf{main}\ e_2$$
$$e_1 = \mathsf{send}(x, p_2, 1)\,;\mathsf{send}(x, p_2, 2)\,;\mathsf{receive}(x, p_2)\,;f(x)$$
$$e_2 = \mathsf{let}\ x' = \mathsf{request}(s, \{p_1, p_2\}, p_1)\ \mathsf{in}\ f(x')$$

$$d_2 = \mathsf{fun}\ g(y) = e_3\ \mathsf{in}\ \mathsf{main}\ e_4$$
$$e_3 = \mathsf{let}\ z_1 = \mathsf{receive}(y, p_1)\ \mathsf{in}\ \mathsf{let}\ z_2 = \mathsf{receive}(y, p_1)\ \mathsf{in}$$
$$\mathsf{send}(y, p_2, z_1 + z_2)\,;g(y)$$
$$e_4 = \mathsf{let}\ y' = \mathsf{accept}(s, p_2)\ \mathsf{in}\ g(y')$$

Figure 1. Code which implements simple math server

3. Problem Exposition and Examples

In Section 3.1 we present a simple maths server, based on an example presented in the literature [6]. We use this example to illustrate the type of error which we want to prevent. In Section 3.2 we present a producer-consumer relationship, and an update to its protocol. We illustrate how we do not need threads to block and synchronise temporally, which would cause overhead when threads are waiting for others. Instead we can coordinate the updates with respect to fulfilling communications responsibilities within a protocol, specifically that each thread performs the same number of runs through the old protocol before migrating to the new protocol.

3.1 Maths Server Example

We present a maths server where a client sends two integers to a server and the server returns their sum. The protocol for this maths server is:

$$G_A = \mu \mathbf{t}.\ \begin{array}{l} p_1 \to p_2 : \langle \textsc{Int} \rangle. \\ p_1 \to p_2 : \langle \textsc{Int} \rangle. \\ p_2 \to p_1 : \langle \textsc{Int} \rangle. \\ \mathbf{t} \end{array} \qquad G_1 = \begin{array}{l} p_1 \to p_2 : \langle \textsc{Int} \rangle. \\ p_1 \to p_2 : \langle \textsc{Int} \rangle \end{array} \qquad G_2 = p_2 \to p_1 : \langle \textsc{Int} \rangle$$

Here p_1 is the client and p_2 is the server: p_1 sends p_2 two integers and p_2 returns a third, their sum. This protocol continues recursively.

We define a system which implements this protocol in Figure 1. We define two program threads, P_1 and P_2, which have thread identifiers i and j and bodies d_1 and d_2 respectively. Thread body d_1 defines a recursive function f, initiates a session s, and passes the channel which it obtains by the session initiation to the function f. Function f is defined so that, using the channel passed to it as argument x, it sends two integers, receives back a third integer, and then recursively calls itself passing itself the channel as its parameter. Thread body d_2 defines a function g, accepts the session being initiated, and calls function g with the channel it obtains. Function g performs the complementary actions to those of f, which are two receives and a send, and then recursively calls itself.

Consider a situation where we want to migrate to a new protocol which permits p_2 to, after having received the two integers, either signal a success and return the result or signal an error and return nothing, and then in both cases to continue recursively. The case of returning an error and no result could be used to deal with exceptions in the server, for example in the case of overflow. We can describe this protocol as follows:

$$G_B = \mu \mathbf{t}.\ \begin{array}{l} G_1. \\ p_2 \to p_1 : \{\mathsf{result}\ :\ G_2.\mathbf{t}, \mathsf{error}\ :\ \mathbf{t}\} \end{array}$$

This extends the previous protocol, as we are sending an additional message, the label which denotes success or error, to the three values we were sending in G_A.

We define a system which implements this new protocol in Figure 2. This is a modification of the code presented in Figure 1, where we have simply replaced the bodies of functions f and g with e_5 and e_7 respectively. Here the select construct permits one thread to decide how the session will continue, either by signalling a result and sending an integer to i and a recursive call, or by signalling an error and a recursive call. For the purposes of this example we choose to signal an error if the first argument is negative.

In our system we do not substitute the function body for function calls at the point when we define the function (which is $\mathsf{fun}\, f(x) = e\, \mathsf{in}\, d$). Instead we provide a heap for each thread which binds the function name to the function body. Thereafter each time we encounter a function call in that thread we replace it with the function body, with the arguments to the function call substituted for the relevant parameter variables.

In order to migrate from a system implementing the first protocol to a system implementing the second all we need to do is replace the function bodies of f and g. Since we store the bodies in the heap, we can simply modify the mapping, and the next time the function is called then the new implementation will be used. This does not replace the inlined code of a function which is running at the time of an update. To safely perform such an update we cannot update the two threads independently; we must perform some coordination of when updates occur. We argue this requirement informally below.

Consider the system $P_1 \| P_2$ defined in Figure 1, and an update which changes the function bodies of f and g from e_1 and e_3 to e_5 and e_7 respectively. To describe the type of error that can occur we first must describe the concept of completing *runs of a protocol* for a given role. If a global session type (protocol) is a loop at top level (e.g. G_A and G_B) then we consider performing the communication actions of the body of the loop as performing one run of that protocol. Different roles will perform different communication actions to perform one run of the protocol: in G_A role p will perform two sends and a receive and role q will perform two receives and a send. Just as roles and threads are separate, runs of a protocol are separate from code control flow: a thread may internally perform two iterations of its main loop to perform one run of the protocol. The notion of a role is specified in the session type itself; to implement this we may need to annotate the code.

We start $P_1 \| P_2$, and we evaluate the threads until both have completed one run of the protocol, resulting in the threads:

$$\langle i, f(k_1) \rangle \| \langle j, g(k_2) \rangle$$

where k_1 and k_2 are the channels which the threads are using to communicate. Since each thread has completed one run of the protocol, at this point two integers have been sent from P_1 to P_2, a third has been sent in response, and all these values have been received. We can then evaluate one step resulting in the threads:

$$\langle i, e_1\{k_1/x\} \rangle \| \langle j, g(k_2) \rangle$$

which performs the function call f in P_1. If at this point we perform the update and then continue to evaluate the function call to g in P_2 then we will have:

$$\langle i, e_1\{k_1/x\} \rangle \| \langle j, e_7\{k_2/y\} \rangle$$

With this configuration, the expressions e_1 and e_7 will attempt to communicate. This, however, will lead to an error since, as we stated earlier, e_7 performs more communications actions than e_5, and hence when e_1 is waiting to receive an integer as the result, e_7 will be attempting to send a label to indicate how it wants to continue the session. Hence we need to perform some form of coordination on when we perform updates, to ensure that such errors do not occur.

$$P_3 \| P_4 \qquad P_3 = \langle i, d_3 \rangle \qquad P_4 = \langle j, d_4 \rangle$$

$$d_3 = \mathsf{fun}\, f(x) = e_5\, \mathsf{in}\, \mathsf{main}\, e_2$$

$$e_5 = \mathsf{send}(x, p_2, 1)\,;\mathsf{send}(x, p_2, 2)\,; e_6$$

$$e_6 = \mathsf{case}(x, p, \{result\ :\ \mathsf{receive}(x, p_2), error\ :\ ()\})\,; f(x)$$

$$e_2 = \mathsf{let}\, x' = \mathsf{request}(s, \{p_1, p_2\}, p_1)\, \mathsf{in}\, f(x')$$

$$d_4 = \mathsf{fun}\, g(y) = e_7\, \mathsf{in}\, \mathsf{main}\, e_4$$

$$e_7 = \mathsf{let}\, z_1 = \mathsf{receive}(y, p_1)\, \mathsf{in}\, \mathsf{let}\, z_2 = \mathsf{receive}(y, p_1)\, \mathsf{in}\, e_8$$

$$e_8 = (\mathsf{if}\, z_1 < 0\ \mathsf{then}\, \mathsf{select}(y, p_1, error)\, \mathsf{else}$$

$$\mathsf{select}(y, p_1, result)\,; \mathsf{send}(y, p_2, z_1 + z_2))\,; g(y)$$

$$e_4 = \mathsf{let}\, y' = \mathsf{accept}(s, p_2)\, \mathsf{in}\, g(y')$$

Figure 2. Code which implements more complex math server

3.2 Producer-Consumer Example

In the maths server example the two threads will only be one run removed from each other, due to the fact that each has to wait to receive from the other. Protocols do not always have such a tightly coupled relationship. Consider a producer-consumer system, defined by the protocol:

$$\begin{aligned} G_C = \mu \mathbf{t}.\quad & p_p \to p_c : \langle \textsc{Int} \rangle. \\ & p_p \to p_c : \langle \textsc{Int} \rangle. \\ & p_p \to p_c : \langle \textsc{Bool} \rangle. \\ & \mathbf{t} \end{aligned}$$

where on each run of the protocol the producer sends the consumer two integers and a boolean. In this situation the producer could, at times, be producing much faster than the consumer is consuming. Consider the situation where the producer has steamed ahead and performed five runs of the protocol, whilst the consumer has only performed the protocol once. For the producer one run of the protocol consists of sending two integers and a boolean, and hence in this scenario it has performed fifteen sends. For the receiver one run of the protocol consists of three receives, and hence in this scenario it has performed three receives. Hence the consumer has an additional twelve values sitting in the queue waiting to be received. At this point the producer could perform an update and migrate to a new protocol:

$$\begin{aligned} G_D = \mu \mathbf{t}.\quad & p_p \to p_c : \langle \textsc{Int} \rangle. \\ & p_p \to p_c : \langle \textsc{Bool} \rangle. \\ & \mathbf{t} \end{aligned}$$

where it only sends one integer and a boolean. It could then go ahead and produce under the new protocol. However, the consumer could not update at this point - it still has four runs worth of messages waiting for it in the queue. Hence it must perform the remaining four runs of the protocol, so that it ends up having performed the correct complementary actions to the producer (specifically the same number of receives, with the same types, in the same order, as the producer has performed sends). After having performed these actions the consumer can perform the update and migrate to the new protocol.

4. Proposed Solution

We use system updates to migrate a system from one protocol to another. A system update includes multiple thread updates which update threads which are involved in the old protocol. Our static analysis, essentially that presented in our previous work [2], is sufficient to guarantee that after a thread has been updated with a well typed update then the thread will perform communications actions according to the new protocol. Hence we can guarantee correctness of migrating individual threads to a new protocol.

Though the individual threads can be guaranteed to cleanly migrate to the new protocol, if the thread updates are applied at the wrong times then we can cause errors. We demonstrate such an error in the evaluation of the maths server example in Section 3.1. Intuitively, the error highlighted in the example is that one thread has begun another run of the old global protocol, whilst the other has migrated to the new global protocol. Hence threads may receive values which they are not expecting or may never be sent the values they are expecting. This violates the fidelity safety property discussed in Section 1 as the protocol would be performing communication actions for which there are no corresponding communication in the protocol. In order to rule out such errors we need to ensure that all threads which are involved in a session migrate to the new session after having completed the same number of runs of the old protocol. After the above static analysis has been used to guarantee that each thread will individually behave correctly, safety becomes a runtime coordination problem. This coordination does not require performing any dynamic typing or analyses, and simply is controlling when thread updates are applied. A system update, whose thread updates are all correct, can always be applied; the runtime coordination will never reject it.

In order to aid reasoning about runs of a protocol we keep a count of how many runs have been completed by each role in that protocol. This count, which we refer to as *run numbers*, denotes how many runs through a protocol have been performed by that role in the protocol. We have to increment the run number of a role every time the thread starts a new run of the global session of which that role is a part. We are not counting loops in the control flow of threads but simply keeping track of the communication actions that have been preformed by a role. Each thread can take part in multiple sessions, including multiple instances of a protocol, and hence each thread can have multiple roles. We augment each thread with a mapping from roles to run numbers.

One possible coordination approach is the naive stop-the-world method, which is the following. After an update has been introduced to the system we define some run number n which is greater than the current run number associated with any of the roles involved in the session being updated. We then block each thread after it has completed n runs. When all threads have blocked then we update the code of all of the threads simultaneously and safely proceed with the new protocol. Whilst this approach would be safe it requires us to block threads to the extent that we are almost stopping the program and restarting. This is at odds with dynamic software update, one of whose primary aims is to provide updates without having to shut down systems.

Instead we can make use of the fact that we use a queueing architecture for message passing. The example in Section 3.2 demonstrates how one thread can safely migrate to the new protocol before another. This is as any messages it sends as part of the new protocol will be put at the back of the queue, and any receives it attempts to do as part of the new protocol will require it to wait for the sender to be using the new protocol. Hence the threads can migrate to the new protocol separately, without requiring synchronising. The point at which each thread can migrate is coordinated as follows.

Each thread has an associated mapping from channels $s.p$ which that thread is using to run numbers n_i (as discussed above). The channel $s.p$ includes the name of the session s and the role p. The mapping is defined as the partial function $\mathsf{runNumber}(p)$. The mapping for thread i is annotated as $\mathsf{runNumber}(p)_i$. We define the *update run number* to be:

$$n = \mathsf{max}(\mathsf{runNumber}(s.p)_i)_{i \in I} + 1$$

where I is the set of thread identifiers of the threads which are being updated and s is the name of the protocol we are updating. Intuitively the update run number is the number of runs through the old protocol a thread must complete after which it must migrate to the new protocol. Hence after a thread has completed protocol run n it applies the update and migrates to the new protocol.

We set the update run number to the maximum run number plus one. To explain this consider again the evaluation of the math server in Section 3.1:

$$\langle i, f(k_1)\rangle \| \langle j, g(k_2)\rangle \rightarrow \langle i, e_1\{k_1/x\}\rangle \| \langle j, g(k_2)\rangle$$

Whilst these threads have both completed the same number of runs of the protocol, thread i has looped and started the code which will perform the next run of the old protocol. If the next evaluation step is applying the update then we can continue to evaluate to:

$$\langle i, e_1\{k_1/x\}\rangle \| \langle j, e_7\{k_2/y\}\rangle$$

and hence have two different versions of the protocol attempting to interact. Since a recursive function call is conceptually an internal action we do not want to reason about it. Instead we assume that, at the point of generating the update run number, the thread with the greatest run number could have started the next run. Hence we set the update run number to one more than the greatest run number.

Once each individual thread has completed n runs of the old protocol it can safely perform the update and migrate to the new protocol. Due to the queueing architecture any messages sent after this migrate will be placed at the back of the queue, and hence other threads which have not yet performed the update should receive exactly the right number and type of messages in their remaining runs of the old protocol.

We cannot define the update run number at compile time as we do not know when the update will be provided to the system. However, as the runtime coordination is concerned only with when, not if, we will apply an update then we retain a purely static analysis.

One implementation issue to note is that in order to calculate the update run number we require a snapshot of the mappings from channels to run numbers for all the threads involved in a protocol being updated. This will necessitate a lock on the threads involved in the protocol, but only for the short amount of time required to calculate the update run number as above. Such a lock is a small operation in comparison to the naive approach which requires locking each thread once it has completed n runs until all threads have reached completed n runs.

5. Related Work

In early work on session typing [9] presents an asynchronous multiparty session type system which includes a limited form of delegation and a progress property to guarantee that a well typed session will not deadlock, in and of itself. The work in [3] builds on [9] to provide full, transparent, delegation as well as a progress guarantee that different well typed sessions will not interfere with each other and cause deadlock. Most work on session types use π calculus style calculi. Some work uses a λ calculus formulation [6, 7] which we used in our analysis to guarantee update safety [2]. All known previous work on session typing restricts function calls to tail calls

in order to simplify the typing. In our update safety analysis we remove this restriction by making of type and effect systems [12].

There are several existing formal analyses for DSU. Early work includes [4] which presents a first-order, simply-typed, call-by-value lambda calculus, with the addition of a module system and an update primitive. An update can change the code of a module, but not its type signature. The programmer can introduce new versions of modules. There is no limit on how many versions of a module the system can have loaded, and code can make explicit which module versions it is willing to use. The update system guarantees type safety of code using different module versions. Following this work [13] presents a C like language, including state, pointers and records (which can be use to implement structures) and facilities to dynamically update code. The granularity of update in this paper is a function; hence the programmer can change the function body (and type signature) of any function and the system will ensure that the updated code is still type safe. This system also permits updates to modify an abstract data type and hence the data items of the type being modified must also be modified so that they conform to the new type. Hence in this system when modifying an abstract data type the programmer must include a function to transform values from the old type to the new type. The work in [11] consists of a similar DSU system to [13], but instead of specifying update points they address the issue of when updates should occur using transactional techniques from database research. In their language the programmer can delineate regions of code inside which she does not wish an update to the code to (visibly) occur. This work is extended to MDSU in a recent paper where one can infer these regions and use a "check in" protocol to only perform the update when all threads are ready to perform it [10]. Experimental results of the delay between updates being introduced and when a suitable update point occurs and discussion about the balance between safety and timeliness are also presented.

Object updating is approached in several papers [1, 5] which present a system which permits objects to be updated in arbitrary ways at different times. Hence it is possible to interact with an older or newer version of an object interface. To deal with this the authors use simulation objects and shadow methods to indicate the effect of older or newer methods on the current object. The authors provide an implementation and informal safety properties.

6. Conclusions and Future Work

We presented an example error which can occur when applying thread updates in an uncoordinated manner. We presented a solution which provides the relevant coordination as to when to apply each update to each thread. This solution does not require any thread to block and wait for others to be ready to perform the update, and hence minimises overhead in the distributed update procedure. This approach does not appear, however, to be applicable to the shared state paradigm unless read and write queues were used for shared state access.

As future work we plan to develop the technical details of the proposed update procedure. This will involve developing mechanisms for threads to signal that they have finished an run of a loop, examining how internal `update` commands interact with the update procedure, developing a type system to analyse such systems, and proving type safety, linearity and fidelity properties similar to those in in our previous work [2].

References

[1] Sameer Ajmani, Barbara Liskov, Liuba Shrira, and Google Inc. Modular software upgrades for distributed systems. In *In Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 452–476, 2006.

[2] Austin Anderson and Julian Rathke. Safe dynamic software update for multi-threaded message-passing systems (draft), 2009. `http://eprints.ecs.soton.ac.uk/18041/`.

[3] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically merged multiparty sessions. Technical report, 2008. `http://www.doc.ic.ac.uk/~yoshida/paper/global_progress.pdf`.

[4] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution. Warsaw, Poland*, 2003.

[5] Rasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 403–417, 2003.

[6] Simon Gay, V. T. Vasconcelos, and Antonio Ravara. Session types for inter-process communication. Technical Report 133, Department of Computing, University of Glasgow, 2003.

[7] Simon Gay and Vasco T. Vasconcelos. Asynchronous functional session types. Technical Report 2007–251, Department of Computing, University of Glasgow, May 2007.

[8] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[9] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, January 2008.

[10] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Not.*, 44(6):13–24, 2009.

[11] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *SIGPLAN Not.*, 43(1):37–49, 2008.

[12] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer-Verlag.

[13] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32 nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 183–194, 2005.