Semantics-Preserving and Incremental Runtime Patching of Real-Time Programs[†]

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Luís Lopes
CRACS/University of Porto
lblopes@dcc.fc.up.pt

Eduardo R. B. Marques
University of Porto
edrdo@dcc.fc.up.pt

Abstract

We propose semantics-preserving and incremental runtime patching of real-time programs as a robust means for reconfiguring hard real-time systems at runtime. We consider programs that describe non-functional aspects of processes such as their timing properties and communication behavior, and give examples written in the Hierarchical Timing Language (HTL). Runtime patching is the process of replacing portions of such programs at runtime by new code. It is semantics-preserving if the switch to the resulting code and the code itself could have been compiled beforehand, had the patch been known. It is incremental if analyzing and generating the code only involves an effort proportional to the size of the patch, not the patched program. This can even be done with system-wide properties such as schedulability by exploiting HTL-specific features.

1. Introduction

Software has the great advantage of being flexible. In fact, for now, it probably remains the single most flexible concept for engineering even the most complex systems. The majority of IT industries exploit that flexibility and sometimes even use it as foundation for their business models. There are important exceptions though. Large portions of the real-time systems industry, in particular, the ones working on mission- and safety-critical applications essentially ignore software-related flexibility. There are good reasons after all. Getting large software systems and, in particular, real-time systems right is still extremely difficult. How can we then even think about modifying such systems while they are running? Clearly, adaptivity is not just a nice-to-have feature, especially in real-time systems where it may give rise to unforeseen application scenarios and software development methodologies. What is even more exciting though is that the essential, enabling technologies may currently be shaping up to make adaptivity of even hard real-time systems a reality.

We believe there are two key ingredients. Adaptivity needs a strong semantical foundation and non-trivial scalability. We need to know what reconfiguration means and how to do it fast, even on large systems. There is a growing research trend towards so-called semantics-preserving execution environments for real-time systems such as the real-time language Giotto [9] and its successors but also other work on synchronous reactive languages [2], which provide notions of composability that go beyond the typical schedulability guarantees of more traditional real-time languages and operating systems. For example, Giotto programs can be modified without changing the relevant properties of the unmodified portions as long as there are sufficient computational resources. Relevant properties are not just schedulability but also task functionality, intertask communication, and I/O times. Nevertheless, checking schedulability and other system-wide properties remains necessary but is often difficult and may limit scalability of reconfiguration attempts. Recent work, however, on incremental schedulability analysis of traditional task models [5] but also language-based models [7], in combination with stronger, semantical notions of composability, may lead to fast, scalable, and semantics-preserving reconfiguration of real-time systems.

In this paper, we propose semantics-preserving and incremental *runtime patching* of real-time programs as a robust means for reconfiguring even large systems at runtime, and give examples written in HTL [7], a Giotto successor. So far, we have only studied the idea conceptually and worked with examples. Our plan is to design and implement runtime patching support in our existing HTL infrastructure [1] and perform experiments with unmanned vehicles in Salzburg [4] and Porto [12]. In Section 2, we give an intuitive overview of our approach. In Section 3, key concepts of HTL are provided, followed by a presentation of an HTL-based runtime patching model in Section 4.

[†]C. M. Kirsch is supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15. L. Lopes is partially supported by project CALLAS from Fundação para a Ciência e Tecnologia (contract PTDC/EIA/71462/2006). E. R. B. Marques is supported by the SFRH/BD/29461/2006 grant from Fundação para a Ciência e Tecnologia.

We would like to thank João Sousa and Raja Sengupta for inspiration in this work and Sebastian Fischmeister for some relevant suggestions and comments.

2. Runtime Patching

We consider programs that describe non-functional aspects of processes such as their timing properties and communication behavior. The syntax tree of such a program P is depicted schematically in the left portion of Fig. 1. The program describes a set of processes as illustrated in the right portion of Fig. 1. We assume that there is a means to identify subprograms of a given program, for instance, by unique path names. The syntax tree in Fig. 1 shows such a path σ to a subprogram S of P. We also assume that there is a homomorphic relationship F between (syntactic) program and (semantical) process composition in the sense that a strict subprogram of a given program can only affect the relevant behavior of a strict subset of the processes described by the program, i.e., $F(P \setminus S + S) = F(P \setminus S) + F(S)$. Fig. 1 indicates that subprogram S only affects the relevant behavior of a strict subset of all processes described by program P. Examples of relevant behavior are process functionality, periodicity, and I/O times while resource consumption such as CPU usage is not. Processes described by S may share resources with processes described by other parts of P and may therefore affect their access to resources. We discuss how to check resource consumption in principle below.

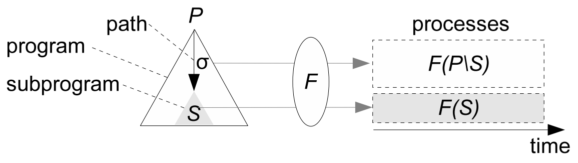

Figure 1. Syntax and semantics

By runtime patching we intuitively mean the process of identifying a subprogram O of a program P by a path σ and then replacing O in P by a new program N, logically instantaneous at runtime, i.e., during the execution of P, resulting in a program P', as shown in Fig. 2. The time instant when the patch takes effect is called the install instant I. Applying a runtime patch may take time and may therefore be started some time before I. However, a runtime patch should only take effect atomically at I, similar to, for example, atomic transactions in databases. Runtime patching enables software adaptivity because patches (σ and N) do not need to be known at compile time, and programs do not need to be stopped for patching. Patch operator implementations may require some form of dynamic loading and linking as well as possibly incremental compilation, unless programs are interpreted.

We do not intend runtime patching to extend the expressiveness of the language in which patched programs are written. In fact, we advocate runtime patching to preserve the exact original language semantics. In other words, there must be a program Q that is syntactically equivalent to P but with O replaced by a conditional expression choosing

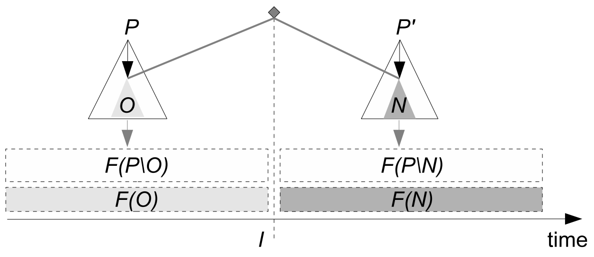

Figure 2. Patching

between O and N as illustrated in Fig. 3. During the execution of Q, the conditional expression, depicted by a box, mimicks runtime patching by switching from O to N exactly at I, i.e., when O in P is patched to become N. Runtime patching is thus a semantics-preserving means to modify programs at runtime in a way that could have been done at compile time, if the timing of the patch (I) and the patch itself (σ and N) had already been known.

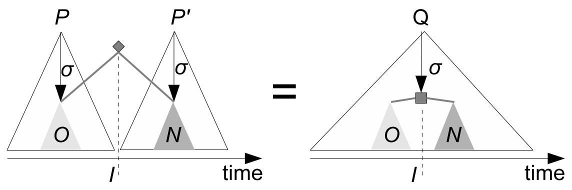

Figure 3. Preserving semantics

Runtime patching involves program analysis and code generation. If a patch requires re-checking program-wide properties such as overall resource consumption, or even full re-compilation, runtime patching may either be limited in scale or may take too long and make the application of the patch ineffective. However, incremental compilation, i.e., incremental program analysis and code generation, may enable fast and scalable runtime patching. With incremental program analysis, checking if the patched program P' is correct, given that the original program P is correct, should only involve an effort proportional to the size of the patch (σ and N) and some context C of the patch but independent of P, as shown in Fig. 4. The size of C should be determined by the size of the patch. Similarly, code generation should be proportional to the size of the new program N, and linking should only involve considering context C.

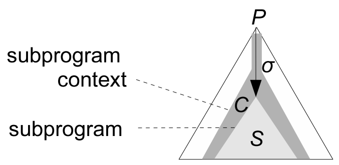

Figure 4. Scalability

Incrementally checking even global properties such as overall resource consumption may also be possible by taking advantage of language properties such as, if P is correct (e.g., resource-compliant) and program N is in some sense compatible with context C, then P' is also correct. This property reduces checking global correctness to checking local compatibility. For example, C may contain an abstract specification implicitly describing a possibly infinite set of concrete programs for which the resulting patched program is guaranteed to be correct, i.e., without re-checking global correctness. Then, checking if a concrete program is compatible with the abstract specification is sufficient for global correctness (but not necessary since there might be concrete programs that are incompatible but for which the patched program is correct anyway).

3. HTL Overview

The Hierarchical Timing Language (HTL) [7] is a coordination language for distributed hard real-time applications. HTL programs specify timing properties and communication behavior of interacting real-time tasks that are potentially distributed across multiple hosts but not task functionality, which is assumed to be implemented in some other language than HTL. Prior to execution, HTL programs are compiled into E code [10], or HE code [8], which supports separate compilation. E and HE code are interpreted in real time by a virtual machine, which uses an EDF scheduler for executing the tasks. The ability to compile parts of HTL programs separately [8] and check their schedulability incrementally [7] is a prerequisite for incremental runtime patching of HTL programs.

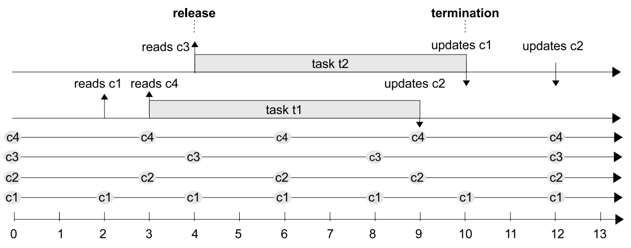

Figure 5. Tasks and communicators

Tasks and communicators. An HTL task is defined by a sequential code procedure with no internal synchronization, a set of input/output variables called *ports*, a period for execution, and a worst-case execution time (WCET). Tasks with different periods interact by exchanging port values through *communicators*, which are timed variables that can be read and written at logical time instants according to the communicators' own periods. This interaction is illustrated in Fig. 5. The periods of interacting tasks must be multiples of the involved communicator periods. Tasks with the same period may interact with other tasks through their ports as long as the tasks read from have completed and the reading tasks have not yet started executing, which gives rise to task precedence constraints.

An HTL task has a *logical execution time* (LET) given by its *release* and *termination* events, which are defined by communicator read and write actions: the release time is the latest time instant for a communicator read and the termination time is the earliest time instant for a communicator write. Fig. 5 illustrates this for tasks `t1` and `t2` and their interaction through communicators `c1` to `c4`. This task model is a generalization of the LET model in Giotto [9], to tasks with input and output ports interacting through communicators. The key advantage of the LET model is that the relevant behavior of LET programs (functionality but also exact I/O times) is preserved across different hardware platforms and software workloads as long as there are sufficient computational resources [9].

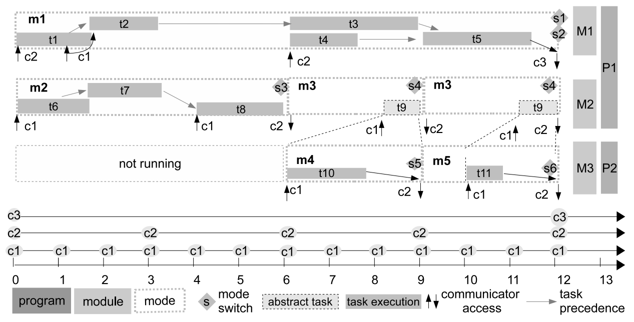

Figure 6. An HTL program

Program structure. Building up on the foundation of tasks and communicators, the other structuring concepts in HTL are modes, modules, programs, and hierarchical program refinement. Fig. 6 gives an overview of their assembly and execution.

A *mode* is a set of tasks with the same period (the mode's period) and an acyclic graph that expresses data flow among input and output ports of the tasks in the mode. A mode's execution equals the logical execution of all its tasks under communicator timing constraints (to interact with tasks external to the mode) but also under task precedence constraints. For example, in Fig. 6, `t2` in mode `m1` is only released after `t1` has completed, even though `t2`'s release time (the access to `c1`) is actually earlier. A *module* is set of modes and a set of boolean predicates called *mode switches* that are evaluated over communicators and ports. At any time, exactly one mode executes within a module, and, at the end of its period, execution is either switched to a different mode in the module or continued in the same mode, according to the evaluation of mode switches. For example, in Fig. 6, in module `M2`, the mode switch `s3` is evaluated at time instant 6 at the end of the execution of mode `m2` and changes execution to mode `m3`. A set of modules and a set of communicator declarations form a *program*, and a program's execution equals the parallel execution of all its modules with the tasks interacting through the program's communicator set. For example, in Fig. 6, program `P1` consists of modules `M1` and `M2` as well as communicators `c1`, `c2`, and `c3`.

Hierarchical program structure is expressed using *refinement* of a mode by an entire program, as shown in Fig. 6 for mode `m3` and program `P2`. A mode being refined, called the *parent mode*, may have declared *abstract tasks* that have no implementation and simply act as schedulability-conservative place-holders for *concrete tasks* in the refinement program conforming to a set of syntactic restrictions [7]. The refinement constraints preserve schedulability and simplify program analysis: if the parent mode is analyzed and asserted as schedulable, then the refinement program is also known to be schedulable. Checking refinement constraints is generally faster and more scalable than checking schedulability and can therefore be done incrementally. The former is linear in the size of the refinement program whereas the latter may be exponential in the size of the refined mode because of higher-level mode switching. We have also studied refinement constraints with so-called logical reliability of communicator updates instead of task schedulability [3] but have not yet considered it in runtime patching.

4. A Runtime Patching Model for HTL

Runtime patching for HTL requires a mechanism to load, analyze, and apply patches at runtime. We propose to use a *patch supervisor* process that monitors a running HTL program and allows its patching. The patch supervisor is not meant to be an HTL entity itself but one that operates on top of it in the sense of a program rewriting other programs in congruence with our principle that syntax and semantics of the patched programs are preserved. The patch supervisor should apply instrumentation in a way that at any time instant the running program is a proper instance of the original language in which it was written. Also, it should be executed at a lower priority than the patched program, so that the real-time performance requirements of the latter are not compromised, and only declare a patch as ready to take effect once all required time-consuming aspects of readying the patch are done. Its typical cycle will be: attend to program patch requests, perform program re-compilation, and apply patches logically instantaneous at time instants that ensure coherent atomic transitions between the original and patched program.

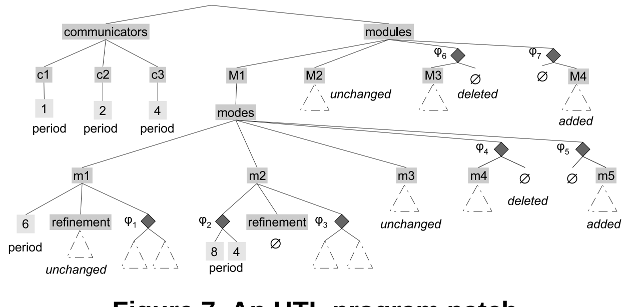

Figure 7. An HTL program patch

Patch specification. Fig. 7 displays an HTL program P in the form of a simplified syntax tree and a patch applied to it yielding program P'. A patch may consist of multiple program transformations, expressed at the syntactic, source-code level, through rewriting of the program's syntax tree. The diamond notation represents a program transformation through patching, with the original subprogram on the left and the new subprogram on the right. The patch shown consists of changes at the mode level for module `M1` (transformations φ_1 to φ_5) and at the module level for the top-level program (φ_6 and φ_7). Patching at the mode level within a module can change an existing mode (φ_1 to φ_3), delete a mode (φ_4), and add a new mode (φ_5). Patching an existing mode may change timing properties like a mode's period, as in φ_2, but also other aspects (e.g., within φ_1 and φ_3) like task precedences, communicator accesses and WCET estimate, as well as functional aspects like task and mode switch implementations. Patching at the module level may remove (φ_6) and add modules (φ_7). Program patching may also be recursive and apply to refinement programs. In constrast to [6], which describes a mechanism for semantics-preserving replacement of real-time program functionality for the Timing Definition Language (TDL), a subset of Giotto and thus of HTL, our approach generalizes to patching concurrent modules and, in particular, modes, besides considering scalability aspects for patching.

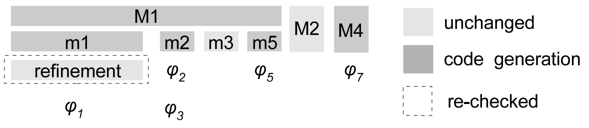

Figure 8. Runtime compilation

Runtime compilation. Compilation of a patched HTL program at runtime must validate the program syntactically to assert the program as valid, analyze the schedulability of the program depending on the type of transformations applied, and re-generate and link code for the changed program parts. Syntactic validation needs to consider only a context composed of the modified parts and their dependencies, which are induced by program refinement and communicator writes that may be performed by pre-existing modules (which could result in race conditions). Code generation for HTL may adopt a separate compilation strategy even down to the level of modes [8]. Thus it is possible to re-generate code only for the modified parts of a patched HTL program.

The subprogram context for syntactic validation and code generation for each transformation in our patching example is illustrated by Fig. 8. Syntactic validation and code generation is required for all changed and added functionality, as shown. Assuming there are no dependencies induced by communicator writes in the example, there is, however, a need to account for the dependencies of program parts changed by program refinement: even though the refinement program for m1 does not require re-compilation, it must nevertheless be re-checked with respect to syntactic refinement constraints to make sure that the patched program is still schedulable.

In general, schedulability analysis, however, may not be scalable if the patch targets top-level specifications. If timing behavior is patched, schedulability analysis is only incremental to changes if the patched program is a refinement but not a top-level program since only refinement constraints preserve schedulability. If a top-level program is in question, as in our example, then schedulability may be asserted through full program analysis but with exponential time complexity in the size of the program, or potentially faster through other incremental schedulability analysis techniques such as in [5], assuming they can be generalized to cover mode switching.

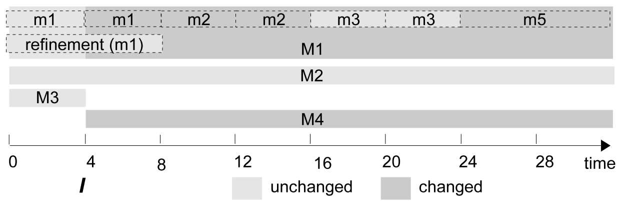

Figure 9. Patched execution

Patched execution. For the patch supervisor to instrument running HTL programs logically instantaneous in a semantics-preserving way, we consider the timing and integrity effects of patching. A runtime patch at the level of modules executing concurrently within an HTL program is constrained by the transformations it involves: (1) modules added by the patch must not be started before the time instant that marks the beginning of the least common multiple of all communicator periods in the module, so that the module has a coherent time origin, (2) modules removed by the patch must terminate execution as soon as the current mode ends execution (the outcome of mode switch evaluation will be ignored), and (3) modules changed by the patch must switch to the patched behavior when execution of the current mode ends, including the evaluation of mode switches which must yield a mode that is defined in the patched program, i.e., one that has not been removed.

The time instants to which patching is constrained by (1) to (3) determine the set of possible install instants for the patch. We assume that activating the patched program takes logically zero time. As discussed before, all time-consuming aspects of runtime compilation complete before the install instant. In the sense of the various aspects surveyed in [11], the runtime patching model we consider is therefore *synchronous*. If the patch involves more than one kind of transformation, the install instant must satisfy all of their timing constraints, i.e., be a valid synchronization point for all transformations. This condition may be relaxed for simultaneous module updates and removals to happen before additions. New modules could start after all module updates and removals have been completed. This mode of operation can be interesting for defining more flexible patching schemes. However, it would imply an interval of logical time for patching, rather than a logical time instant, and require additional schedulability analysis, as in *asynchronous* patching [11]. In any case, a patch supervisor has the flexibility of applying different transformations that may be part of a set of patches at different appropriate time instants, so the above constraints may not be too restrictive.

Fig. 9 illustrates patched execution for our example. The patch is applied logically at time instant 4 in line with the constraints stated above. Time instant 4 is the least common multiple of all communicator periods (1, 2, 4 for c1, c2, and c3 in Fig. 7), so that M4 can be started at that time assuming that mode execution for M1 (modified) and M3 (deleted) properly terminates. The patched execution within module M1 shows changed components and sample mode switching behavior. When patching has completed, execution is switched from m1 (according to the specification of old code for m1) to the patched version of itself. Any other switch would also be valid as long as the target mode is defined in the patched program (m2, m3, or m5). A mode switch to m4 at time instant 4 would invalidate the patching operation since the patch specifies m4 to be deleted.

References

[1] J. Auerbach, D. Bacon, D. Iercan, C. Kirsch, V. Rajan, H. Röck, and R. Trummer. Java takes flight: Time-portable real-time programming with Exotasks. In *Proc. LCTES*, 2007.

[2] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM TECS*, February 2008.

[3] K. Chatterjee, A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, C. Pinello, and A. Sangiovanni-Vincentelli. Logical reliability of interacting real-time tasks. In *Proc. DATE*, 2008.

[4] S. Craciunas, C. Kirsch, H. Röck, and R. Trummer. The JAviator: A high-payload quadrotor UAV with high-level programming capabilities. In *Proc. AIAA GNC*, 2008.

[5] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hierarchical real-time components. In *Proc. EMSOFT*, 2006.

[6] S. Fischmeister and K. Winkler. Non-blocking deterministic replacement of functionality, timing, and data-flow for hard real-time systems at runtime. In *Proc. ECRTS*, July 2005.

[7] A. Ghosal, T. Henzinger, D. Iercan, C. Kirsch, and A. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Proc. EMSOFT*, 2006.

[8] A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, and A. Sangiovanni-Vincentelli. Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code. In *Online Proc. APGES*, 2007.

[9] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc. of the IEEE*, January 2003.

[10] T. Henzinger and C. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. PLDI*, 2002.

[11] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *RTS*, Springer, 2004.

[12] Seascout LAUV. http://whale.fe.up.pt/seascout.