Runtime Programming through Model-Preserving, Scalable Runtime Patches

Christoph M. Kirsch[1], Luís Lopes[2], Eduardo R.B. Marques[2], and Ana Sokolova[1]

[1] Department of Computer Sciences, University of Salzburg.

[2] CRACS/INESC-Porto LA, Faculdade de Ciências, Universidade do Porto.

1 Introduction

We propose a methodology for flexible software design, runtime programming, by means of incremental software modifications at runtime. Runtime programming acknowledges that software designs are often incomplete, and require the flexibility of change, e.g., fixing bugs or introducing new features, without disruption of their service. This flexibility is much needed for critical software that generally needs to handle uncertainty, e.g. cloud computing or cyber-physical systems, due to dynamic requirements of composition, service, or performance. Runtime modifications should be allowed recurrently, and, thus, be handled as a common case of system functionality in predictable and efficient manner, with proper understanding of inherent functional and non-functional aspects. The work in many diverse research communities with related concerns typically tends to take a partial and domain-specific view of the problem, hence comprehensive and general methodologies are in order.

In this extended abstract, we make a summary of the runtime programming proposal of [4]. The work follows up on a preliminary formulation of runtime programming [3], and work on modular compilation of real-time programs [2].

2 Runtime programming through runtime patches

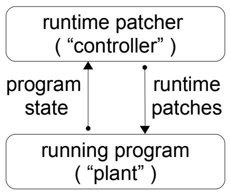


Fig. 1. Runtime programming [4].

The runtime programming abstraction is illustrated in Fig. 1. A program (bottom) is subject at runtime to recurrent incremental modifications, called runtime patches, by an external program, a runtime patcher (top). A runtime patch determines a switch between two program specifications and states of these programs, by replacing a component in the source program. Runtime patches are applied by the patcher in congruence with program state and the (evolving) program does not stop, instead it flows with any introduced runtime patches. An obvious analogy exists with the “controller-plant” formulation of control theory: the evolving program is the “plant”, the patcher is the “controller”, and runtime patches define the “control”.

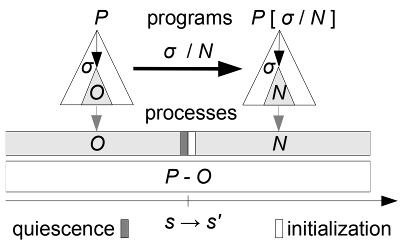


Fig. 2. A runtime patch [4].

The transformation defined by a runtime patch, illustrated in Fig. 2, has well-defined syntactic and semantic effects.

Syntactically, a patch σ/N over program P defines the substitution of a component at “program path” σ, $O = P[\sigma]$, by component N, yielding program $P\ [\sigma/N]$. Strict component addition or removals are special instances of this effect, when O or N are undefined respectively.

Semantically, the patch defines an instantaneous switch from a state s of P to a state s' of $P\ [\sigma/N]$ such that: the processes of O terminate, according to a notion of “quiescence” in place that expresses graceful conditions for doing so (e.g., inactivity, or atomic instants that marks the end of a component’s “transaction”); the processes of N start in a valid initial state; and the state of processes of other components $P - O$ is unaffected ($s[P - O] = s'[P - O]$).

Thus, runtime programming assumes only a simple abstraction of component-based software, comprised of a modular relation between (the syntax of) components and (the semantics of) processes, plus built-in notions of initialization and quiescence.

3 Model preservation

We consider that runtime programming should be model-preserving. i.e., preserve the programming model in place for programs in terms of program syntax, semantics, and correctness properties. More precisely, model preservation is the guarantee that, in a runtime programming system, a proper program is running at all times, and a corresponding state for that program is observed that complies with correct operation. The point is avoiding an “ad-hoc”, disruptive nature for runtime patches, and relying on no particular abstraction level other than the one already in place for programs.

The concept of a runtime patch, described above, already provides relatively strong model-preserving provisions. A runtime patch defines an atomic switch, and observes proper quiescence, initialization, and isolation of replaced, new, and unchanged components, respectively. This means that in a runtime programming system, a proper program is running at all times, and a corresponding state of that program is observed, with a safe continuous flow observed upon patch effect. Hence there are no transient “meta-programs” or “meta-behavior” that are alien to the syntax or semantics of programs.

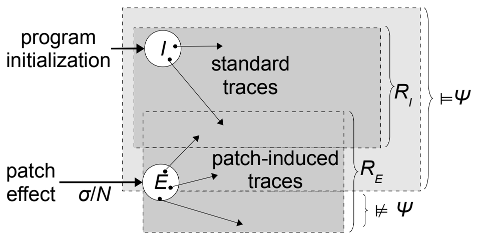
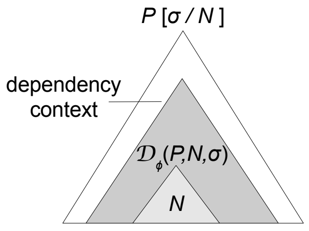


Fig. 3. Model preservation [4].

Additionally, we should expect a program that is "started" through patch effect behaves correctly. That is, its execution should conform to any particular properties of correctness for the program, were it to execute from scratch. The problem, however, illustrated in Fig. 3, is that a runtime patch σ/N over a given program P defines a partial, "live" initialization of $P\ [\sigma/N]$, rather than a whole-program initialization. So it could happen that "patch-induced traces", those with states R_E resulting from effect E of σ/N over the flow of P, do not conform to a certain expected property ψ of correctness that holds for "standard traces" of $P\ [\sigma/N]$, those with states R_I resulting from overall initial conditions I. We say the patch is model-preserving if and only if R_E is a subset of the satisfiability state space of the correctness property ψ. Checking model preservation should be an integral part of the process of patch compilation, described next.

4 Scalability

The complexity of a runtime programming system should ideally scale with the "size" of runtime patches. Such complexity comes from patch compilation, the set of procedures required to verify and integrate a patch, such as checking a model-preserving nature for patches, and other aspects like code generation or re-linking. If patch compilation does not scale in the general case, the practicality of runtime programming is compromised.



$$I(\phi) = (\psi_\phi, \mathcal{D}_\phi, C_\phi)$$

$$\frac{P \models \phi \quad \mathcal{D}_\phi(P,N,\sigma) \models \psi_\phi}{P\ [\sigma/N] \models \phi}$$

$$C_\phi(P,N,\sigma) = \mathrm{O}\left(\mathcal{D}_\phi(P,N,\sigma) \models \psi_\phi\right)$$

Fig. 4. Scalability [4].

To characterize scalability of patch compilation, we propose the incremental compilation framework illustrated in Fig. 4. The idea is that for a patch σ/N over program P, compilation should be incremental to that of P, and scale appropriately in proportion to the extent of the patch, as determined by P, σ and N. For each aspect of compilation ϕ (e.g. code generation), an incremental strategy should be defined, $I(\phi) = (\psi_\phi, \mathcal{D}_\phi, C_\phi)$

with the following rationale. For a patch σ/N over (the previously compiled) P, ϕ should be dealt with for $P\,[\sigma/N]$ by some incremental effort ψ_ϕ over a dependency context of components $\mathcal{D}_\phi(P, N, \sigma)$. The complexity of checking ψ_ϕ over $\mathcal{D}_\phi(P, N, \sigma)$ by some algorithm is expressed by $C_\phi(P, N, \sigma) = \mathrm{O}\left(\mathcal{D}_\phi(P, N, \sigma) \models \psi_\phi\right)$, which we call the compilation cost. In Fig. 4, it is shown (left) that the dependency context $\mathcal{D}_\phi(P, N, \sigma)$ is a set of components within $P\,[\sigma/N]$, and, additionally, it may also include the "old" component $O = P\,[\sigma]$. The inference rule on the right of the figure expresses the incrementality in compilation: under the assumption that P already verifies ϕ, it is just required to verify ψ_ϕ over $\mathcal{D}_\phi(P, N, \sigma)$.

This formulation inherently characterizes incremental compilation and its scalability, in the size (dependency context) and time (compilation cost) dimensions. Scalability can be broken in one of the dimensions, e.g., when a patch requires the full program as context, or if the incremental effort is intractable, as measured by the compilation cost. A good degree of scalability corresponds to a small dependency context, and a tractable incremental effort. To achieve it, the choice of compilation strategy may in some cases represent a loss of precision. A strategy that scales well, and covers the more general cases of valid patches reasonably, will be preferable to one that is more exact, but scales poorly. This is important in particular when we are faced with the well-known "state explosion problem" incurred by an exact analysis.

5 Ongoing work

In [4] we provide a detailed description and formalization of runtime programming, corresponding to the general overview given here. Additionally, we put the formulation in perspective with a case-study instantiation of runtime programming for a component-based language for distributed real-time systems, the Hierarchical Timing Language (HTL) [2, 1]. In earlier work [3], some of these ideas and HTL runtime patching were discussed in preliminary short form but mainly considering the specific context of real-time systems and HTL. An incremental compilation framework was proposed for HTL already in [2], which we generalized now for component-based systems in the context of runtime programming.

Acknowledgements. This work has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the Austrian Science Funds P18913-N15 and V00125, and Fundação para a Ciência e Tecnologia funds PTDC/EIA/71462/2006 and SFRH/BD/29461/2006.

References

1. Ghosal, A., Henzinger, T., Iercan, D., Kirsch, C., Sangiovanni-Vincentelli, A.: A hierarchical coordination language for interacting real-time tasks. In: Proc. International Conference on Embedded Software (EMSOFT). pp. 132–141. ACM (2006)
2. Henzinger, T., Kirsch, C., Marques, E., Sokolova, A.: Distributed, modular HTL. In: Proc. Real-Time Systems Symposium (RTSS). pp. 171–180. IEEE (2009)
3. Kirsch, C., Lopes, L., Marques, E.: Semantics-Preserving and Incremental Runtime Patching of Real-Time Programs. In: Online Proc. Workshop on Adaptive and Reconfigurable Embedded Systems (APRES). pp. 3–7. ARTIST Network of Excellence (2008)
4. Kirsch, C., Lopes, L., Marques, E., Sokolova, A.: Runtime Programming through Model-Preserving, Scalable Runtime Patches. Tech. Rep. 2010-08, Department of Computer Sciences, University of Salzburg (2010)