UNIVERSITÄT SALZBURG

Runtime Programming through Model-Preserving, Scalable Runtime Patches

Christoph M. Kirsch Luís Lopes[a] Eduardo R. B. Marques[a]
Ana Sokolova

[a]CRACS / INESC-LA, University of Porto

Technical Report 2010-08 December 2010

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
www.cosy.sbg.ac.at

Technical Report Series

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, Austria
Email: {ck,anas}@cs.uni-salzburg.at

[2] CRACS / INESC-Porto LA,
Faculdade de Ciências,
Universidade do Porto
Email: {lblopes,edrdo}@cs.uni-salzburg.at

Abstract. We consider a methodology for flexible software design, runtime programming, defined by recurrent, incremental software modifications to a program at runtime, called runtime patches. The principles we consider for runtime programming are model preservation, ensuring the model in place for programs is preserved whilst allowing for runtime patches, and scalability, understanding how program compilation can appropriately scale in proportion to the change induced by a runtime patch. In accordance, a runtime patch is proposed as a model-preserving operation, working as a correct instantaneous transition between programs, and one that can be incrementally compiled for scalability. We put the formulation in perspective through a case-study instantiation over a language for distributed hard real-time systems, the Hierarchical Timing Language (HTL).

1 Introduction

We propose a methodology for flexible software design, runtime programming, defined by recurrent, incremental modifications to a program at runtime. Runtime programming acknowledges that software designs are often incomplete, and must therefore evolve over time, but should do so in a well-defined, systematic manner, ideally with little disruption of service. This type of flexibility is much needed for critical software in the presence of uncertainty, due to dynamic requirements of composition, service, or performance – consider for example cloud computing or cyber-physical systems. Runtime modifications should be allowed intensively, 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. Related work in many diverse research communities – e.g.: programming languages [29]; operating systems [1]; databases [10]; large-scale web servers [4]; real-time control systems [37]; or sensor networks [27, 25] – and in industry – e.g., the OMG OSGi [32] and IEC 61499 [21] standards – typically tends to take a partial view of the problem in domain-specific manner, hence comprehensive and general methodologies are in order.

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

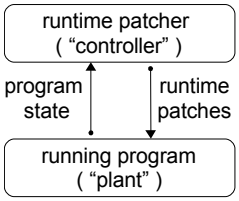


Fig. 1. Runtime programming.

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, but 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”.

We consider two key principles for runtime programming: model preservation and scalability.

The model preservation principle is that runtime programming should preserve the model in place for programs, i.e., the base model used for programming in the first place. The behavior of a runtime programming system should be explained alone by the base model, and comply with a correct interpretation of that model. We reason that there are two major aspects to it. First, it should be ensured in any event that a proper program is running, and that a corresponding state for that program is observed, rather than transient “meta-programs” and “meta-behavior” induced by patches. Secondly, a safe continuous flow should be observed upon patch effect, and correct program operation should be ensured afterwards, rather than a disruptive transition. The runtime patching operation we propose has these aspects in mind. A runtime patch specifies a transformation of program specification (syntax), that affects the behaviour of processes described by the program (semantics) in instantaneous form. The effect of a runtime patch observes requirements that guarantee quiescent termination of replaced functionality, proper activation of the new one, and isolation of effect for other unchanged components. Plus, program behavior is required to be correct after patch effect, rather than deviate from it. The assumption is a simple abstraction of component-based software, comprised by a modular relation between (the syntax of) components in a program and (the semantics of) the processes they describe, plus built-in notions of initialization and quiescence.

The scalability principle is that the complexity of a runtime programming system should scale with the “size” of runtime patches. The complexity stems from what we call patch compilation, the set of procedures required to verify and integrate a patch, such as checking patch correctness, ensuring patch feasibility (conditions for eventual effect), and other aspects like code generation. If the process does not scale in the general case – for instance if a full program re-compilation is required per patch – the practicality of runtime patching will be compromised. Instead, it is desirable that patch compilation proceeds incrementally, taking at most a “dependency context” of the por-

tion of the program affected by the patch. With this in mind, we propose a patch compilation framework, that defines how to conduct patch compilation incrementally, and inherently characterizes its scalability.

Our contribution is a characterization of runtime programming in the terms above. It comprises three parts: (1) assumptions on a component-based program model, that serves as base for runtime programming; (2) the definition of runtime programming over that component-based program model, with runtime patching at its core; and (3), a characterization of incremental patch compilation. Runtime programming is then put in perspective with a case-study, taking in context a component-based language for real-time distributed control systems, the Hierarchical Timing Language (HTL) [17, 15].

The remainder of this paper is structured as follows. Section 2 puts forward our general formulation of runtime programming, and our HTL case-study instantiates it in Section 3. Section 4 defines runtime programming in formal terms, generally, and then specifically with regard to the HTL instantiation. We should note that Section 4 is important for matters of preciseness and detail of formulation, but can otherwise be skipped for the purpose of understanding the general runtime programming formulation or the HTL case-study. Related work is discussed in Section 5, and Section 6 concludes.

2 Runtime programming

2.1 Program model assumptions

We take as assumption a simple model for component-based programs with two main features. The first is a clean modular relation between the specification of a program, its syntax, and the behavior of that program, its semantics. The other is the existence of built-in notions for graceful activation and deactivation of functionality, initialization and quiescence. The point is to define a general design pattern for programs that allows for incremental modifications at runtime with a well-defined functional effect.

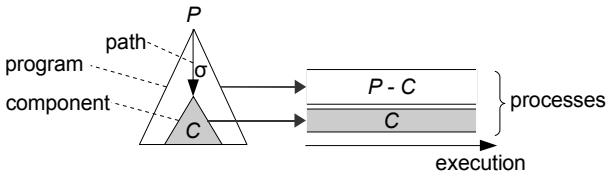


Fig. 2. Program model assumptions – syntax and semantics.

Syntax and semantics. We consider a program is specified by composition of (syntactic) components, and its execution is defined by sets of (semantic) processes. This is shown in Fig. 2 for program P (left), and corresponding processes (right). We assume that the inner components in the syntax tree of a given program, or more generally of any given component, are identified by unique path names. In the figure a path σ is

shown in P, identifying a component $C = P[\sigma]$. The paths are an abstraction of hierarchical composition at the syntactic level.

We further assume components and processes are modularly related in the sense that a component of a given program can only affect the relevant behavior of a strict subset of the processes described by the program. In Fig. 2, C and its sub-components are only meant to specify the functional behavior of a strict subset of processes of P in isolated manner, i.e., $\texttt{processes}(P) = \texttt{processes}(P - C) \cup \texttt{processes}(C)$. Examples of relevant functionality may comprise data processing, process interaction, or I/O. The processes of C and $P - C$ may however interfere and be correlated in non-functional aspects, e.g., resource consumption such as processor or network usage. Functional and non-functional aspects should be addressed by program compilation, comprising program verification and code generation, operating over the syntax of programs, as discussed later.

Initialization and quiescence. To model graceful activation and deactivation of components, we assume that built-in notions termed initialization and quiescence are in place for process computation. This is illustrated in Fig. 3 for some component C with three associated processes, p_1 to p_3. Initial states merely define valid conditions for processes to start. Quiescent states reflect the completion of logically indivisible operations in consistent form, or process idleness with no side effects. Both notions generalize to the execution of overall components, as shown for C in Fig. 3: initial and quiescent states of a component's execution are those such that all corresponding processes are in an initial state or quiescent state, respectively.

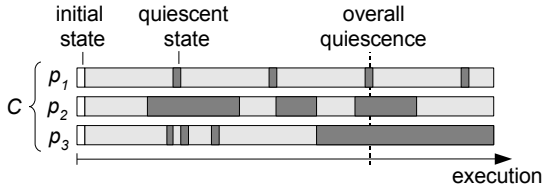


Fig. 3. Program model assumptions – initialization and quiescence.

We should note quiescence is a fundamental notion to reason on non-disruptive runtime programming. It instantiates in several forms in work that considers runtime modifications to a system, e.g., detecting non-active kernel functions in operating systems [1], reaching annotated program points in multi-threaded programs [29], or the dynamic update model for distributed systems in [23].

2.2 Formulation of runtime programming

We define runtime programming over the component-based model assumptions. Recalling the scheme of Fig. 1, runtime programming comprises a runtime patcher inducing incremental software modifications, runtime patches, over a running program. We now formulate the two concepts of runtime patch and runtime patcher.

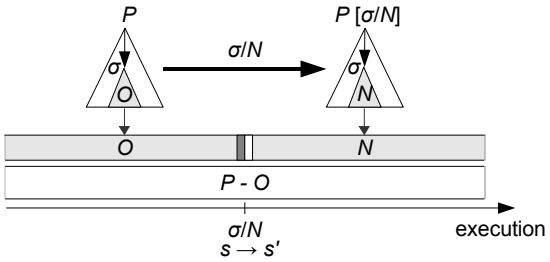


Fig. 4. A runtime patch.

Runtime patch. A runtime patch is illustrated in Fig. 4. A patch σ/N is defined by a path σ, and a component N. The application of σ/N over a program P defines the runtime replacement of component O at program path σ in P by N, yielding subsequent execution of program $P\,[\sigma/N]$.

Syntactically, $P\,[\sigma/N]$ is a program in which N, the "new component", has path σ, and replaces $O = P[\sigma]$, the "old component". All other paths (components) outside the scope of σ are preserved from P to $P\,[\sigma/N]$. The strict addition (resp. removal) of components is a special instance of this syntactic effect, respectively, when O (resp. N) is undefined. For a program P, if a program $P\,[\sigma/N]$ exists in these conditions, we say patch σ/N is well-formed for P.

Semantically, the effect of a well-formed patch σ/N over P is an atomic switch $s \xrightarrow{\sigma/N} s'$ between a state s of P, and a state s' of $P\,[\sigma/N]$, that observes the following requirements:

– **Quiescence** – s is a quiescent state for all processes of O, i.e., O is guaranteed to terminate gracefully.

– **Initialization** – s' defines a valid initial state for processes of N, i.e., N initiates properly.

– **Isolation** – s' preserves the state of s with regard to processes associated with the set of components $P - O$, i.e., the execution of unchanged components is not affected.

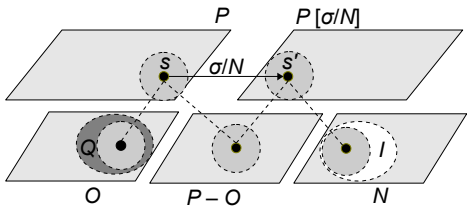


Fig. 5. Patch effect – state space of components.

In Fig. 5, semantic patch effect is depicted in terms of the state space of P, $P\,[\sigma/N]$, and projection of that state space for involved components. For patch effect, O must

enter a quiescence zone Q, N must be able to start from a valid initialization zone I, and the state of unchanged components $P - O$ must remain the same. We say σ/N is feasible if the execution of P guarantees eventual semantic effect of σ/N, i.e., from every possible execution state s_0 of P, a state s is always eventually reached such that $s \xrightarrow{\sigma/N} s'$ for some state s' of $P\,[\sigma/N]$.

Runtime patcher. A runtime patcher is considered as an abstract entity that has the ability to observe (as input) the syntactic structure and the semantic state of a currently executing program, and define (as output) runtime patches that modify that program and its state, in adherence to the constraints put forward for patching. With regard to the nature of a runtime patcher, our intention is to merely reason at this very abstract level, without considering, so to say, how the patcher comes up with the patches, or its design. Naturally, however, actual requirements of a runtime programming system can be elaborate, in the same vein of reconfigurable "live systems" [24, 42].

The notions of runtime patcher and runtime patching define the possible executions of a runtime programming system, in the form illustrated in Fig. 6. The figure shows that, starting from an initial configuration where program P_1 is active, a patcher $\mathcal{P}$ has the ability of inducing patches $\sigma_1/N_1, \sigma_2/N_2, \ldots$ over program execution. The resulting program sequence is

$$P_1, P_2 = P_1\,[\sigma_1/N_1], P_3 = P_2\,[\sigma_2/N_2], \ldots,$$

and the resulting program state sequence is

$$s_1, \ \ldots\ , s'_1, s_2, \ \ldots\ , s'_2, s_3, \ \ldots\ , s'_3, \ \ldots,$$

such that intermediate state sequences $s_i, \ \ldots\ , s'_i$ are defined by the semantics of P_i, and $s'_i \xrightarrow{\sigma_i/N_i} s_{i+1}$.

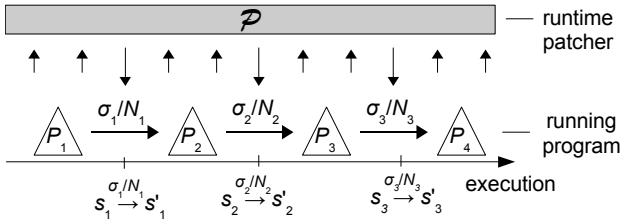


Fig. 6. Runtime patcher and program execution.

Model preservation. The runtime patching operation we propose partially fulfills the requirements put forward in the introduction for model preservation. A patch defines an instantaneous switch between two programs, thus its effect can be explained alone by the programming model in place. Moreover, patch effect defines a safe continuous flow between programs, by observing proper quiescence, initialization, and isolation, for functionality that is removed, introduced, or unchanged, respectively. We must however enforce the additional requirement for model preservation that correct program behavior

is ensured after patch effect. Well-formedness of a patch σ/N over a program P, the fact that $P[\sigma/N]$ is a proper program, and patch feasibility conditions, do not guarantee that to be the case necessarily. The issue is that a runtime patch defines a different (a partial) initialization of (otherwise a priori correct) $P[\sigma/N]$, attending only to the immediate effect of the patch, not to subsequent program behavior, which may potentially deviate from correctness.

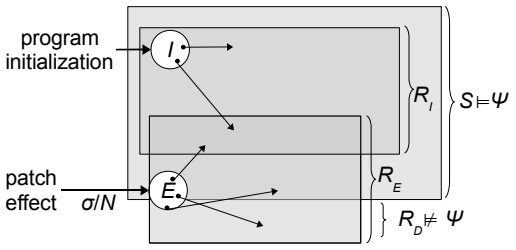


Fig. 7. Deviation from correctness after patch effect.

The problem is illustrated in Fig. 7. The state space of a program $P[\sigma/N]$ is shown, considering execution starting from overall initial conditions I, or starting through effect of a patch σ/N from E, as a "continuation" of the execution of P. Patch-induced program traces, those starting from E, and defining reachable state space R_E, may differ, transiently or even in the long term, from standard program traces, those starting from I, and defining reachable state space R_I. The figure shows that the state space S of satisfiability of a certain property of correctness ψ, includes all states of standard program traces ($R_I \subseteq S$), but that this may not necessarily hold for patch-induced traces. A model-preserving patch should preclude the existence of deviant behavior $R_D \subseteq R_E$, shown at bottom in Fig. 7, where ψ does not hold.

This approach does not necessarily require any special relation between patch-induced traces and program traces, such as, e.g., the condition for valid runtime patches in [16], requiring convergence of patch-induced traces to standard program operation. In this sense, though, particularly if liveness properties are at stake, it may be expectable that patch-induced traces and standard traces converge, or are very closely related. We should note also that there might be properties of correctness that are difficult or impossible to analyze beforehand, and runtime patching obviously complicates that task. This problem has been addressed with redundancy and fault isolation methodologies [37, 11, 8], that counter for deviation of correctness after patch effect. We do not consider this type of methodologies, but their use is not ruled out at the program level by our component-based model assumptions, e.g., an analytic redundancy relation between components [37] can be in place.

Patching scope and decomposition. The requirements put forward for runtime patching can possibly be quite sensitive in terms of possible scope attainable by runtime patching. After all, for effect of a patch, a possibly delicate synchronization of quiescence, valid initialization, and isolation of effect is required for different components.

Moreover, compliance with correctness after patch effect is also required. Consider for instance large patches that are infeasible, because they replace components that quiesce in unsynchronized manner, or that are too complex to verify w.r.t. correctness after patch effect.

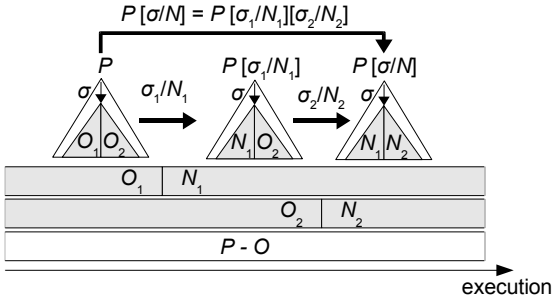


Fig. 8. Patch decomposition.

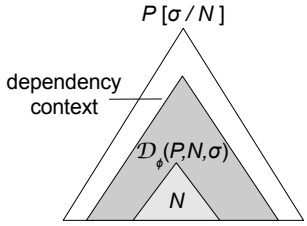
It should be the case that the set of model-preserving patches is much smaller than the set of well-formed patches, those that merely encode a valid syntactic transformation. This should not be seen as a limitation. The correct perspective is that the runtime patching operation can work as a sound inductive case to deal with otherwise invalid program patches. To handle these, we can consider the decomposition in smaller model-preserving sub-patches that proceed in several sequential steps, such that the overall final effect corresponds to the intended transformation. The idea is illustrated in Fig. 8. A patch σ/N is shown applied to a program P, by decomposition in smaller patches σ_1/N_1 and σ_2/N_2. The requirement is that $P\,[\sigma/N] = P\,[\sigma_1/N_1][\sigma_2/N_2]$, and that σ_1/N_1 and σ_2/N_2 can operate over P and $P\,[\sigma_1/N_1]$, respectively.

Patch decomposition can reflect a number of aspects related to the programming model in place, or design choices for better performance w.r.t. metrics of choice. For instance, the order of patches in a decomposition can express component dependencies, e.g., in Fig. 8, it can be that σ_2/N_2 does not proceed first, due to a "dependency" of N_1 by N_2. Factors such as promptness, availability, or correctness, may determine the pattern of progressive software change, e.g, as in mode change protocols for real-time systems [35], or upgrade protocols in large-scale web servers [4]. Another general use of decomposition may be to break down a component replacement into a component removal, followed later, after some downtime at the replacement path, by an addition – consider $N_1 = \perp, N_2 = N$ in Fig. 8, where $\perp$ stands for an undefined component. The downtime may be necessary for several reasons: synchrony of effect with other patches, matching initialization requirements of the new component, or time consuming state-transfer from old to new component (e.g., [5, 38]).

2.3 Patch compilation

Patch compilation is the process of verifying and integrating a runtime patch in a runtime programming system. Verification of a runtime patch over a given program needs to establish the conditions for model-preservation (well formedness, feasibility, correctness after effect) formulated previously. For these, functional or non-functional properties of correctness (e.g., deadlock-freedom, resource consumption) are to be taken in consideration, along with program analysis w.r.t. patch effect. Patch integration may comprise aspects such as code generation, or relinking. Overall, patch compilation may make use of an array of specialized techniques, e.g., see [1, 39, 6, 28–30, 27, 2]. Our interest, though, is not to characterize particular techniques for patch compilation, but instead to propose a general methodology for their scalable implementation.

The key observation we make is that, in a runtime programming environment, a patch changes a "previously compiled" program, hence compilation should be able to proceed incrementally. To characterize incremental patch compilation, we consider a base framework originally defined in [17], and generalize it for our abstraction of component-based software.



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

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

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

Fig. 9. Incremental patch compilation.

The proposal is illustrated in Fig. 9. The idea is that patch compilation should consist of an incremental effort operating over the dependency context of components related to a patch. Per each compilation aspect ϕ, say for instance code generation, the dependency context of a patch σ/N over P, $\mathcal{D}_\phi(P, N, \sigma)$, shown left in the figure, identifies the portion in $P\,[\sigma/N]$ that needs to be accounted for to deal with ϕ incrementally, taking also, optionally, $O = P[\sigma]$ in consideration. The incremental effort seeks to establish a property φ_ϕ over that dependency context through some algorithm. This is expressed by the inference rule shown right in the figure: ϕ is dealt with for $P\,[\sigma/N]$ if φ_ϕ is considered over $\mathcal{D}_\phi(P, N, \sigma)$, and also under the assumption that P has been previously compiled w.r.t. ϕ. The inherent time complexity of this incremental compilation effort is in turn expressed by $C_\phi(P, N, \sigma) = O\left(\mathcal{D}_\phi(P, N, \sigma) \models \varphi_\phi\right)$, called the compilation cost – we abuse notation in the sense that the complexity at stake relates to the algorithm in place to verify φ_ϕ. We call $I(\phi) = (\varphi_\phi, \mathcal{D}_\phi, C_\phi)$ an incremental compilation strategy for ϕ.

The formulation above 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., if a patch requires the full program as context, or if the compilation cost has intractable complexity. A good degree of scalability corresponds to a small dependency context, and a tractable incremental compilation effort.

Our methodology is tightly related to matters of modular compilation, component composition, trade-offs between precision and performance in the compilation of component-based systems, and in particular the well known state-explosion problem in this context. In this sense we share concerns with [3, 41, 26, 14]. By our formulation in Fig. 9, $P \models \phi \wedge \mathcal{D}_\phi(P, N, \sigma) \models \varphi_\phi$ is a sufficiency criteria for $P\,[\sigma/N] \models \phi$, hence exact compilation may not be expressed. The dependency context and compilation cost depend on the choice of φ_ϕ, which can be seen as a degree of freedom in patch compilation, representing the balance between precision and effort.

3 Case study

We put our proposal in perspective considering runtime programming over the HTL language [17, 15]. HTL is a component-based coordination language for real-time distributed control systems. HTL programs are expressed as the composition of syntactic components, under constructs for concurrency, choice, and hierarchical refinement. The execution of components corresponds to the semantical composition of real-time tasks in a platform, with guarantees of predictable timing behavior. We begin by providing an overview of the language. We then describe how runtime programming can be defined over it, considering how the language fits our programming model assumptions, and how runtime patches can be defined and compiled. As an assessment of practicality, even though we have not devised an implementation of runtime programming over HTL, a running example of a real-world application is discussed throughout the text.

3.1 HTL overview

Example. We consider a running example, concerning an HTL application for a three-tank system (3TS) [20]. The 3TS, depicted in Fig. 10, consists of three tanks, Tank_1, Tank_2, and Tank_3. Each tank has an evacuation tap, Tap_1 to Tap_3, and there are two tank inter-connecting taps, $\text{Tap}_{1,3}$ and $\text{Tap}_{2,3}$. Two pumps Pump_1 and Pump_2 control the flow of water into Tank_1 and Tank_2, with the aim of maintaining the water level in the tanks, both in the case of water leaks through the tank's taps, or in their absence. For pump control, a proportional (P) controller is used in the absence of leaks, and two proportional-integrative (PI) controllers are used when there are leaks, one with slow integration speed for an estimated low control error, the other with faster integration speed. To control the 3TS system, an HTL program has been implemented [20], and some videos demonstrating it at work can be found in [19]. In this paper, we consider a small adaptation of the original 3TS program, by letting P controllers mentioned above run at 1 Hz, and PI controllers run at 2Hz, rather than a fixed frequency of 2 Hz for all control in the original program.

In Fig. 11, the syntactic structure of the adapted `3TS` program (left) and a possible execution of it (right) are shown. The `3TS` program runs on three hosts. Two hosts

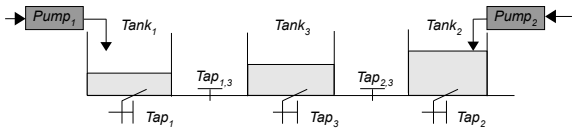


Fig. 10. Three-tank system [20].

have direct access to the two pumps, respectively, and the remaining host serves as a monitoring interface to an operator. Overall, Fig. 11 illustrates that an HTL program is a hierarchical tree-like structure composed of other components called modules and modes, which can in turn define inner programs through a relation of hierarchical refinement. The top-level program `3TS` consists of three concurrently running modules `Pump1`, `Monitor`, and `Pump2`, each mapped to one of the mentioned hosts. Each module is organized in modes, which describe switchable configurations of operation in the module, with each mode defining the invocation of a set of real-time task invocations over a time period, some of which can be abstract placeholders for hierarchical refinement. In the example, the pump control modules `Pump1` and `Pump2` have similar structure. Each of the modules has two modes, corresponding to P-control and PI-control modes of the 3TS. The PI-control mode is further refined by a program that defines the "slow"-PI and "fast"-PI control modes.

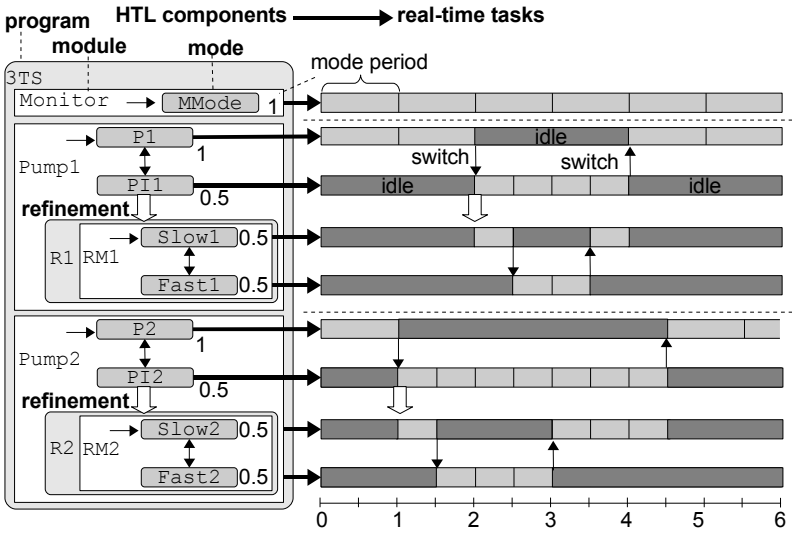


Fig. 11. An HTL program for the 3TS (adaptation from [20]).

HTL components. In more detail, HTL components are defined and relate to each other as follows:

– A program defines a set of modules that execute concurrently. A top-level (root) program can be distributed module-wise across different hosts in a network (as in the 3TS example), and defines a set of global interaction variables called communicators, described below.
– A module is defined by a set of modes, with one mode identified as start mode (e.g., in Fig. 11, `P1` for module `Pump1`), and some mode switching logic expressed by conditions over communicator variables. A module executes by activating one mode at a time, beginning with the start mode, and evaluates mode switching logic to define the next mode to execute in sequence (e.g., in Fig. 11, module `Pump1` initiates with mode `P1` at time 0, switches to `PI1` at time 2, and then back to `P1` at time 4).
– A mode defines any number of real-time tasks, and their invocation over a fixed time period, called the mode's period (e.g., in Fig. 11, mode periods are 1 or 0.5). Tasks in a mode are expressed as insulated I/O functional blocks with no internal synchronization, and computation specified using an external programming language, such as C or Java. The end of a mode's period is consistent with graceful termination of all computation of tasks in the mode, and defines the time for evaluating mode switching at the upper level of the parent module.
– The hierarchical refinement of a mode by an entire program, called the mode's refinement program, is enabled in case some tasks in a mode are abstract placeholders with no implementation. Refinement does not add expressiveness to HTL, as it is possible to transform any hierarchical program into a "flat" program without refinement. It does however offer the flexibility of hierarchical encapsulation, which in turn can leverage the effort in compiling a program. The refinement program must provide concrete task implementations in modes of equal periods, or even abstract tasks again, if refinement is nested. Other refinement constraints are also enforced, with the general intent of preserving key properties of the parent mode's specification, such as schedulability of computation. Refinement programs are active when their parent mode is also active, executing concurrently, and in time-synchronized form at mode switching instants. In Fig. 11, one can see mode `PI1` and its refinement `R1` activated together in interval $(2, 4)$, and, likewise, `PI2` and `R2` in interval $(1, 4.5)$.
– HTL components interact with each other and external software in the runtime envi-

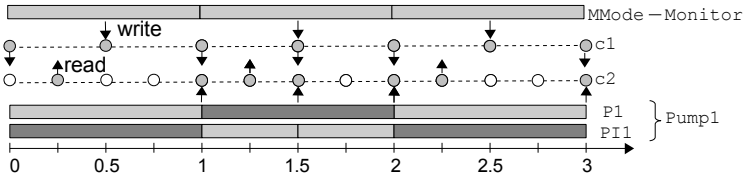


Fig. 12. HTL communicator interaction.

ronment through communicators. A communicator is a global top-level program variable that has an associated period for access by real-time tasks in a mode, meaning it can only be logically read or written at times that are multiples of its period. Valid interaction is also defined by the absence of races in communicator writes, i.e., no two

components should write to the same communicator at the same logical time. A communicator value persists in between writes, that need not occur at every communicator period or synchronize with reads, and is broadcasted upon update over the network. A communicator that is updated externally is called a sensor communicator, and the values of sensors over time are called the program inputs. Fig. 12 depicts a possible communicator interaction between the modules `Monitor` and `Pump1` of the 3TS program, using communicators `c1` and `c2` with periods 0.5 and 0.25, respectively. Tasks in mode `MMode` within module `Monitor` read `c2`, and write to `c1`. The tasks of modes `P1` and `PI1` in module `Pump1` write to `c2`, and read from `c1`.

Time-determinism. The desired key property for correctness of an HTL program is time-determinism. A program is time-deterministic if for every timed sequence of inputs (sensor communicator values over time) the program always yields a unique timed sequence of outputs (the values of all other communicators over time). Time-determinism is ensured for a program by the absence of races in communicator updates in the program's specification, plus, attending to the constraints of a given platform, schedulability of task computations per host, and schedulability of network transmissions for communicator updates. A time-deterministic program can thus guarantee predictable functionality, and also one that is portable over any given platform with sufficient computational resources. The verification of time-determinism is part of the HTL compilation process, described below.

3.2 Runtime programming over HTL

Program model assumptions. HTL adheres to the program model assumptions of Section 2.1 with the following traits:
– HTL syntax and semantics are modularly related, since each component describes an isolated set of processes in the form of real-time tasks. Moreover, each component is uniquely named at each syntactic level [17], e.g., all modules in a program have distinct names within a program, which allows for a trivial definition of a component path scheme (e.g., the path of `P1` in Fig. 10 could be something like `3TS.Pump1.P1`).
– A notion of component initialization is also in place. It takes form in two essential aspects. First, each module must initialize from its start mode. This applies to top-level modules, when activated from an overall top-level program initial state, and refinement-level modules, whenever their parent mode becomes active after a period of idleness (e.g., `R1` and `PI1` at time 2 in Fig. 11). Secondly, initialization requires that the beginning of a mode's period is harmonic with the timeline of all communicators it accesses, or that other modes in the same module do, as illustrated in Fig. 12.
– HTL component quiescence can be seen expressed by intervals of idleness, when a component's execution has no side effects (e.g., in Fig. 11, $(0, 2)$ and $(4, 6)$ for `PI1` and all its sub-components) or atomic instants of mode switching in all modes of a component (e.g., `PI1` again every 0.5 seconds in $(2, 4)$). Quiescent is a guaranteed eventually for modes, modules, or refinement level programs. The same does not hold for top-level programs, as they can perform mode switching in non-synchronized form indefinitely – e.g., in Fig. 11, after time 3.5, `Pump1` and `Pump2` start switching at times 4, 5, ..., and 3.5, 4.5, ... , respectively – except in very specific cases – e.g., if all modes in a program

have equal periods, or if every module has just one mode allowing for synchronization on an hyperperiod.

Runtime programming formulation. We consider runtime patches over HTL programs with model-preserving instantaneous effect, or through decomposition, by letting patches proceeding in decomposed form as a sequence of smaller model-preserving patches.

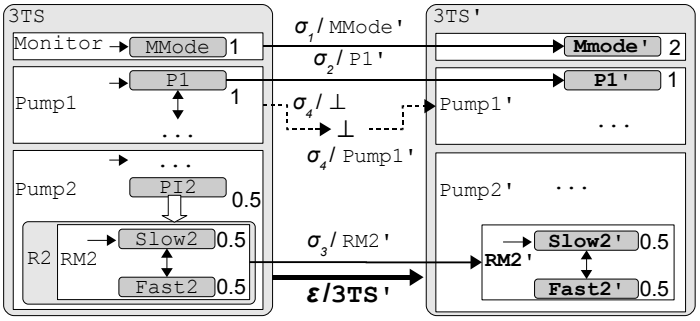
We let a model-preserving patch define the replacement of modes, modules, and refinement programs, or the addition or removal of modules. The reason for not considering mode or refinement program addition is that they are infeasible by definition, without being subsumed, respectively, by a change on the containing module's mode switching logic, and by a patch to the parent mode associated to a refinement program. We also only consider top-level program patches by decomposition, since quiescence is not in the general case a guaranteed behavior for top-level programs – hence, little expressiveness is lost. Additionally, we consider the simplification that a patch does not affect the communicator set of a program. This avoids a number of technicalities without loss of expressiveness: a patch that changes communicators can be seen as equivalent, through communicator "renaming", to a patch between programs with the same communicator set.

The design and implementation of an actual runtime patcher are not a core concern in this paper. For illustrative purposes, one can conceive for example that a runtime patcher is some piece of software interfacing with a distributed HTL runtime system [15, 14] that is extended for runtime code instrumentation. In the following discussion, however, the main issue is to highlight possible choices of actuation by a runtime patcher, specifically with regard to patch decomposition policies and respective trade-offs.

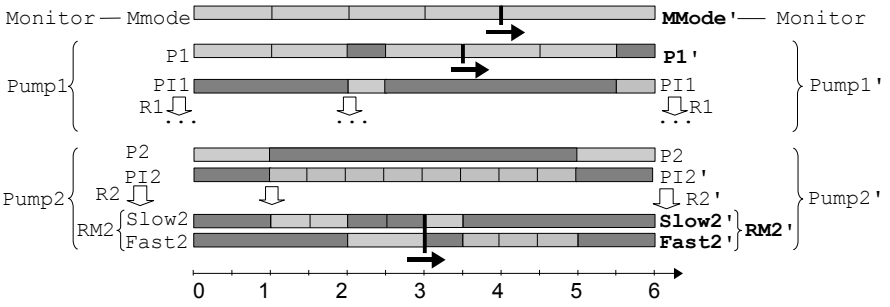
3TS patch example. To illustrate runtime programming over HTL, we consider a patch over the 3TS example, depicted in Fig. 13. The figure shows the syntactic changes defined by the patch (13a), and two possible effects by decomposition (13b and 13c).

The example patch, $\varepsilon/\texttt{3TS}'$, syntactically changes `3TS` by replacing top-level modes `MMode` ($\sigma_1/\texttt{MMode}'$) and `PI1` ($\sigma_2/\texttt{P1}'$), plus refinement module `RM2` ($\sigma_3/\texttt{RM2}'$). The patch over `MMode` yields new mode `MMode'` with a different period (2 rather than 1), and it is assumed that `MMode'` has an initialization constraint due to hypothetical communicator access definitions, such that it can only start at time instants multiples of 2. The patch over `RM2` is assumed to replace all component modes and change the module's switching logic. For simplicity, the example is artificial by considering that the functionality in each 3TS pumps changes differently. A more "natural" patch would consider the same type of changes on both pump modules.

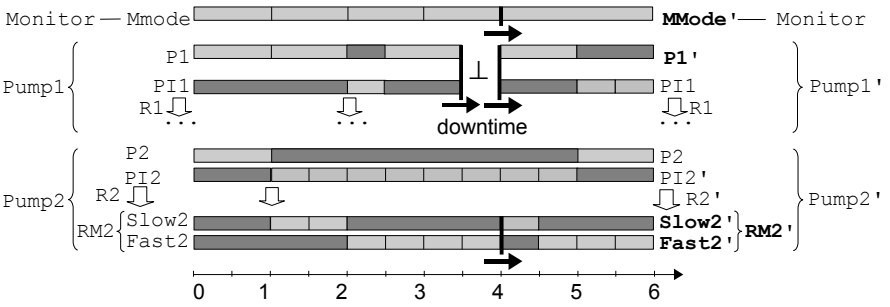
The $\varepsilon/\texttt{3TS}'$ patch of Fig. 13a needs to proceed in decomposed form. A possible decomposition results from considering the strict syntactic changes over paths σ_1 to σ_3. But other decompositions can be considered flexibly. As illustrated in Fig. 13a, for instance, patch $\sigma_2/\texttt{P1}'$ can be subsumed by the module replacement $\sigma_4/\texttt{Pump1}'$, and in turn the latter can be decomposed further into a removal of `Pump1` ($\sigma_4/\perp$), followed by the actual addition of `Pump1'` ($\sigma_4/\texttt{Pump1}'$). Fig. 13b and Fig. 13c depicts sample semantic effects of the two decompositions. A patcher is assumed to induce patches after time 3 in the $(0, 6)$ timeline. In Fig. 13b, the patcher proceeds by applying patches on



(a) Syntactic change



(b) $3TS' = 3TS\,[\sigma_3/RM2'][\sigma_2/P1'][\sigma_1/MMode']$



(c) $3TS' = 3TS\,[\sigma_4/\perp][\sigma_1/MMode'][\sigma_3/RM2'][\sigma_4/Pump1']$

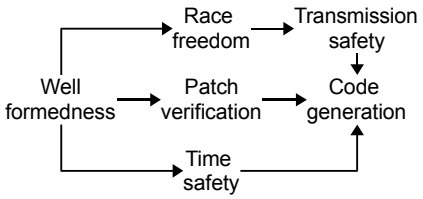
Fig. 13. 3TS program patch.

paths σ_1 to σ_3, opportunistically as soon as possible. The patch over MMode is delayed until time 4, due to the initialization constraint on MMode′. The patch over P1 is also delayed until time 3.5, since P1 is not quiescent at time 3. Only the patch over RM2 can proceed at time 3 immediately. In Fig. 13c, for the same execution until time 3, the patcher proceeds by delaying all components replacements until time 4, and, to obtain synchrony of effect, inducing downtime on program path σ_4 in interval $(3.5, 4)$ by removal of Pump1 .

The effects of the two decompositions in Fig. 13 are representative of asynchronous and synchronous real-time mode changes [35], as the first (Fig. 13b) expresses an asynchronous mix of "old" and "new tasks" in a transient period, and the second (Fig. 13c) does not. As such, they represent a trade-off between several competing factors: promptness in patch effect, downtime in affected program paths, and also of compilation effort, discussed below.

3.3 HTL Patch compilation

We describe HTL patch compilation as an extension to the work of [17], where a staged compilation process for HTL has been described, plus associated incremental compilation strategies. The HTL compilation process comprises several compilation aspects, and precedences between them, in the manner shown in Fig. 14a. The additional aspect of patch verification is considered in the figure, extending standard compilation. The associated incremental compilation strategies are summarized in the table of Fig. 14b, in terms of the the dependency context – $\mathcal{D}_\phi(P, N, \sigma)$ – and compilation cost – $C_\phi(P, N, \sigma)$ – measures of Section 2, considering a patch σ/N over a program P, inducing component replacement or addition. The simpler case of component removal is characterized separately below. A distinction is also made in Fig. 14b for some compilation aspects, regarding whether N is a top-level component, or a refinement-level component.



(a) Compilation aspects.

ϕ	$\mathcal{D}_\phi(P, N, \sigma)$		$C_\phi(P, N, \sigma)$	
	top	ref.	top	ref.
Well formedness	N		Linear	
Race freedom	$P\ [\sigma/N]$	$\emptyset$	Linear	Void
Time Safety			Exponential	
Transmission safety			Linear	
Patch verification	O and N		Linear	
Code generation	N		Linear	

(b) Incremental compilation strategies.

Fig. 14. HTL patch compilation (extension of [17]).

As shown in Fig. 14, incremental HTL compilation proceeds first by checking compliance with syntactic constraints (well formedness). Checking for well-formedness requires taking a dependency context of N alone, and has a compilation effort linear to the size of N. Time-determinism is then ensured by checking the absence of races in communicator writes (race freedom), schedulability of computation (time safety), and schedulability of network broadcasts for communicator value propagation (transmission safety). The later aspects are preserved from P by well formedness of N alone, if N is a refinement-level component. Otherwise, they need to take the entire program as dependency context, but generally proceed with a linear effort. The exception is verification of time safety, that induces an exponential time effort, assuming an EDF scheduling scheme [17]. At the end of the compilation process, code can be generated for the "hierarchical" E-machine [14], for N alone and with a linear effort.

Beyond patch verification, discussed below, a few refinements to HTL compilation should be necessary. Code generation [14] should counter for runtime relinking, but with the same incremental characterization in principle. Runtime patching also raises the issue of component removal. Attending to [17], well-formedness of $P\ [\sigma/\bot]$ is verifiable with a linear effort taking $O = P[\sigma]$ in context, and time-determinism is preserved from P. Also, no code generation is required for $P - O$, but unlinking the code of O should be necessary (in principle, again a linear effort over O).

Patch verification. In line with the formulation of Section 2, the validation of a model-preserving patch comprises checking for patch well-formedness (validation of syntactic patch effect), patch feasibility (ensuring eventual semantic effect of the patch), and compliance with correctness (time-determinism) after patch effect.

Patch well-formedness is subsumed by standard incremental compilation of HTL programs. So is the issue of compliance with time-determinism after patch effect. Potential deviation from time-determinism could result from patch effect, as concurrent tasks in different modes execute together in a different manner from an execution from scratch. But the compilation strategies in place for time-determinism (race freedom, time safety, and transmission safety) already consider an over-approximated state space defined by all potential mode switching combinations in different modules [17]. Hence time-determinism can be ensured in all possible executions after patch effect. The reason for the over-approximation is that a precise analysis is not possible, since whether a particular mode switch will occur or not is undecidable.

Thus, patch verification merely requires checking patch feasibility. For this, we must deal with the initialization, quiescence, and isolation requirements of runtime patching, and ensure they eventually hold in the execution of a program P for semantic effect of a given patch σ/N. As indicated in Fig. 14b for the patch verification aspect, this takes $O = P[\sigma]$ and N as context and has a linear time effort, when N is a top-level component. It is otherwise ensured by well-formedness alone for a refinement-level component N. The reasons for this are explained in more precise manner in Section 4.4, but the rationale is as follows. The quiescence requirement always holds, since we restrained a priori model-preserving patches from being defined over top-level programs. So does the isolation requirement, by virtue of isolation of state for HTL components. The initialization requirement requires that O always quiesces at instants that are con-

sistent startup times for top-level N, something verifiable in linear-time through simple constraints over the periods of communicators accessed by both components.

Scalability. Overall, scalability in HTL patch compilation is compromised by the verification of time safety for patches that affect top-level components. This is of course a serious burden, but a number of established techniques can be in principle used to overcome the problem, referred to in Section 5.

The 3TS patching example of Fig. 13 illustrates this, along with compilation trade-offs for choices of patch decomposition. The decomposition of Fig. 13b defines three programs, respectively resulting from patches at three distinct program paths, and at separate times (3, 3.5, and 4). The two patches over `MMode` and `P1`, operate at the program top-level, thus require re-checking the program for all compilation aspects, and in particular time safety, which induces an unscalable compilation effort. The other patch over `RM2`, a refinement-level component, merely requires checking for well-formedness and generating code in isolation for the component, with a linear effort for both. The compilation effort in the decomposition of Fig. 13c is also hindered by the top-level time safety check, but reduced essentially to one program, the "final" `3TS′` resulting from effect of three simultaneous patches. The module removal (`Pump1`/$\perp$) anticipating those patches can be handled in comparatively lightweight manner, with linear compilation cost. The two decompositions instantiate a known trade-off in real-time mode changes [35], obtaining higher promptness at the cost of extra checks for correctness in a period when "old" and "new" tasks mix.

4 Formalization

In this section we provide a definition of runtime programming in formal terms. The formalization addresses matters of detail and preciseness in regard to the characterization in Section 2, but otherwise expresses the same concepts. Also as in Section 2, we proceed by characterizing the program model assumptions, the formulation of runtime programming, and incremental patch compilation, in this order. We also illustrate the HTL instantiation of the formalization, in connection to the HTL syntax and semantics of [17].

4.1 Program model assumptions

Syntax. We assume a syntax for programs, expressed by: a domain of components `Components`; a domain of programs `Programs` $\subseteq$ `Components`; a set Σ of path symbols; and a mapping

$$[\,] : \texttt{Components} \times \Sigma^* \rightarrow \texttt{Components} \cup \{\perp\},$$

called the path function.

A path is a sequence of symbols $\sigma = \alpha_1 ... \alpha_n \in \Sigma^*, n \geq 0$, with the empty sequence (defined for $n = 0$) denoted ε. We let $\sigma_1\sigma_2$ denote the concatenation of paths σ_1 and σ_2, and $\sigma_1 \preceq \sigma_2$ denote that path σ_1 prefixes or equals σ_2. When $\sigma_1 \preceq \sigma_2$, we say σ_2 is in the scope of σ_1. If two paths σ_1 and σ_2 do not prefix one another – $\sigma_1 \not\preceq \sigma_2, \sigma_2 \not\preceq \sigma_1$ – we say they are concurrent paths, and write $\sigma_1 \parallel \sigma_2$.

For a component C, we write $C[\sigma] = C_0$ if $[\,](C, \sigma) = C_0$. If $C_0 \neq \bot$, we write $\sigma \in \texttt{paths}(C)$, and we say that component C_0 is declared in path σ. If $C_0 = \bot$ we say that path σ is undefined in C. We denote $C - C_0$ to be the set of components in C with paths concurrent to the path of a sub-component C_0 of C. We impose three constraints on the relation between paths and components, in line with the intuition in Fig. 2. For every $C \in \texttt{Components}$ we require that:

$$C[\varepsilon] = C \tag{1}$$

$$\forall \sigma_1 \in \texttt{paths}(C), \sigma_2 \in \Sigma^*, C[\sigma_1\sigma_2] = C[\sigma_1][\sigma_2] \tag{2}$$

$$\forall \sigma_1, \sigma_2 : \sigma_1 \parallel \sigma_2, \nexists \sigma : C[\sigma_1\sigma] = C[\sigma_2\sigma] \neq \bot \tag{3}$$

That is, the empty path always identifies the root-level component in context (1), paths compose (2), and components in concurrent paths are distinct (3).

Semantics. We assume a domain of processes $\texttt{Processes}$, and a mapping of components to sets of processes

$$\texttt{processes} : \texttt{Components} \to 2^{\texttt{Processes}},$$

such that for all components C, and $\sigma_1, \sigma_2 \in \texttt{paths}(C)$:

$$\sigma_1 \preceq \sigma_2 \Rightarrow \texttt{processes}(C[\sigma_1]) \supseteq \texttt{processes}(C[\sigma_2]) \tag{4}$$

$$\sigma_1 \parallel \sigma_2 \Rightarrow \texttt{processes}(C[\sigma_1]) \cap \texttt{processes}(C[\sigma_2]) = \emptyset \tag{5}$$

The constraints mean that processes of a component must include those of their sub-components (4), and that components in concurrent paths have disjoint process sets (5), as illustrated previously in Fig. 2.

Per component C, we take the semantics of $\texttt{processes}(C)$ to be expressed by a tuple

$$(\texttt{states}(C), \texttt{init}(C), \xrightarrow{C}, \texttt{q}(C))$$

where: $\texttt{states}(C)$ is a set of states; $\texttt{init}(C) \subseteq \texttt{states}(C)$ is a non-empty subset of initial states; $\xrightarrow{C}$ is a left-total binary relation on $\texttt{states}(C)$ called the successor relation; and $\texttt{q}(C)$ is a subset of $\texttt{states}(C) \times \texttt{processes}(C)$, called the quiescence relation. The formulation of component semantics is basically a kind of Kripke structure, of common use to model abstract semantics (e.g., see [7] for reference).

We say state s of component C is a quiescent state for process p if $(s, p) \in \texttt{q}(C)$, and that s is overall quiescent for C if it is quiescent for all processes in $\texttt{processes}(C)$, denoted by $s \in \texttt{qstates}(C)$. For $(s, s') \in \xrightarrow{C}$, we write $s \xrightarrow{C} s'$, and say s' is a successor of s. A trace of C is defined as a sequence of states of the form

$$s_0, s_1, s_2, \ldots : s_0 \in \texttt{init}(C) \wedge \forall i \geq 0, s_i \xrightarrow{C} s_{i+1}. \tag{6}$$

We wish to enforce that the semantics of a component expresses the composition of any sub-components it contains, by some simple semantics-preserving constraints over a notion of state projection. If a component C contains a sub-component C_0, a notion of projection in C_0 is required to be in place, for every state s of C, $s\lfloor C_0 \rfloor \in \texttt{states}(C_0)$, such that:

$$s \in \texttt{init}(C) \Rightarrow s\lfloor C_0 \rfloor \in \texttt{init}(C_0) \tag{7}$$

$$\begin{bmatrix} p \in \texttt{processes}(C_0) \\ \wedge \\ (s,p) \in \texttt{q}(C) \end{bmatrix} \Rightarrow (s\lfloor C_0 \rfloor, p) \in \texttt{q}(C_0) \tag{8}$$

$$s \xrightarrow{C} s' \Rightarrow \left[s\lfloor C_0 \rfloor = s'\lfloor C_0 \rfloor \vee s\lfloor C_0 \rfloor \xrightarrow{C_0} s'\lfloor C_0 \rfloor \right] \tag{9}$$

Thus, a projection is required to preserve initialization (7), quiescence (8), and successor relation (9). Note that by (9), a successor of a state in C either maintains the projection within a sub-component, or corresponds to a successor of the projection, which is a general abstraction for any type of process composition, e.g., forms of interleaving, synchronization, etc. Under these constraints, a trace of C in the form of (6) always projects onto a trace of a sub-component C_0 of C, defined by $s_{n_0}\lfloor C_0 \rfloor, s_{n_1}\lfloor C_0 \rfloor, s_{n_2}\lfloor C_0 \rfloor, \ldots$, where $n_0 = 0$ and, for $i \geq 0$, $n_{i+1} = \min\{k > n_i : s_{n_i} \xrightarrow{C} s_k\}$.

Finally, to reason on program correctness, we assume a set Ψ defining correctness properties to which programs comply. A correctness property is a logical predicate over program traces. We require for $\psi \in \Psi$ that ψ holds for every trace of a program.

4.2 Runtime programming formulation

Runtime patching. A patch is a pair (σ, N), denoted σ/N, where σ is a path, and N is a component or an undefined value $\perp$.

Let P be program, σ/N be a patch, and $O = P[\sigma]$.

We say σ/N is well-formed for P, if there is a program, denoted $P\,[\sigma/N]$, such that:

$$P\,[\sigma/N]\,[\sigma] = N \tag{10}$$

$$\forall \sigma_0 : \sigma_0 \parallel \sigma,\ \ P\,[\sigma/N]\,[\sigma_0] = P[\sigma_0] \tag{11}$$

The above conditions expresses the syntactic effect of a patch, which is to replace O by N in P (10), while preserving all other paths (11). The patch is called a component removal if $N = \perp$, a component addition if $O = \perp$, and a component replacement otherwise ($O, N \neq \perp$).

We say σ/N has a defined semantic effect $s \xrightarrow{\sigma/N} s'$ between $s \in \texttt{states}(P)$ and $s' \in \texttt{states}(P\,[\sigma/N])$ under the following conditions:

$$O \neq \perp \Rightarrow s\lfloor O \rfloor \in \texttt{qstates}(O) \tag{12}$$

$$N \neq \perp \Rightarrow s'\lfloor N \rfloor \in \texttt{init}(N) \tag{13}$$

$$C \in P - O \Rightarrow s\lfloor C \rfloor = s'\lfloor C \rfloor \tag{14}$$

Thus, semantic effect requires that: state s is quiescent for the processes of O (12), except if O is undefined (the component addition sub-case); state s' defines a valid initial state for N (13), except if N is undefined (component removal); and state s' preserves the state of processes of components in $P - O$ from s (14).

We say a well-formed patch σ/N over P is feasible over P, if the execution of P always eventually leads to conditions for semantic effect of σ/N i.e., more formally,

$$\forall s_0 \in \texttt{States}(P),\ \exists s : s_0 \xrightarrow{P} \ldots \xrightarrow{P} s \wedge s \xrightarrow{\sigma/N} \cdot \tag{15}$$

Finally, we say a feasible patch σ/N over P is model-preserving, and write $\sigma/N \in \texttt{patches}(P)$, if every sequence of the form

$$s_0, s_1, s_2, \ldots : \cdot \xrightarrow{\sigma/N} s_0 \wedge \forall i \geq 0, s_i \xrightarrow{P\,[\sigma/N]} s_{i+1}$$

called a patch-induced trace for σ/N over P, verifies the properties of correctness Ψ in place for program traces, i.e., for $\psi \in \Psi$, ψ must also hold for patch-induced traces.

The notion of patch decomposition can be subsequently formalized as follows. Given patch σ/N over program P, we say σ/N is decomposable, and write $\sigma/N \in \texttt{dpatches}(P)$, if there are patches σ_1/N_1 and σ_2/N_2, such that (in line with Fig. 8):

$$P\,[\sigma_1/N_1][\sigma_2/N_2] = P\,[\sigma/N] \tag{16}$$

$$\sigma_1/N_1 \in \texttt{patches}(P) \cup \texttt{dpatches}(P) \tag{17}$$

$$\sigma_2/N_2 \in \texttt{patches}(P\,[\sigma_1/N_1]) \tag{18}$$

$$\sigma_1 \parallel \sigma_2 \vee (\sigma_1 = \sigma_2 = \sigma \wedge N_1 = \bot) \tag{19}$$

That is, σ_1/N_1 and σ_2/N_2, applied in this order: yield the same syntactic effect of $P[\sigma/N]$ (16); are decomposable or model-preserving (17) and model-preserving respectively (18); and affect concurrent program paths (19), unless the decomposition reflects a component replacement broken-down in the component's removal followed by its addition ($N_1 = \sigma/\bot$ over P, $N_2 = \sigma/N$ over $P\,[\sigma/\bot]$). Note that conditions (16) (18), (19) imply a finite recursion over (17).

Runtime patcher. A runtime patcher $\mathcal{P}$ is a tuple

$$(\texttt{states}(\mathcal{P}), \texttt{init}(\mathcal{P}), \xrightarrow{\mathcal{P}}, \Longrightarrow^{\mathcal{P}})$$

defined by: a state domain $\texttt{states}(\mathcal{P})$; an initial state domain $\texttt{init}(\mathcal{P}) \subseteq \texttt{states}(\mathcal{P})$; a labelled successor relation

$$\hat{s} \xrightarrow[P,s]{\mathcal{P}} \hat{s}'$$

that can be defined for $\hat{s}, \hat{s}' \in \texttt{states}(\mathcal{P})$, $P \in \texttt{Programs}$, and $s \in \texttt{states}(P)$, indicating $\hat{s}'$ is a successor of $\hat{s}$ by observation of state s of program P; and a patching relation

$$s \Longrightarrow^{\mathcal{P}}_{P,\sigma/N,\hat{s}} s'$$

that can be defined for $\hat{s} \in \texttt{states}(\mathcal{P})$, $P \in \texttt{Programs}$, $\sigma/N \in \texttt{patches}(P)$, and $s, s' : s \xrightarrow{\sigma/N} s'$, indicating patcher state $\hat{s}$ can induce the stated patch effect over program P.

Runtime programming system. We express the execution of a runtime programming system, through the notions of runtime programming state and runtime programming trace, as follows.

A runtime programming state has the form $r = (P, s, \hat{s})$, where P is a program, s is a state of P, and $\hat{s}$ is a state of a patcher $\mathcal{P}$. A runtime programming trace is defined as a sequence of runtime programming states $r_0, r_1, r_2, \ldots$ where $r_0 \in \texttt{Programs} \times$

$\texttt{init}(P_0) \times \texttt{init}(\mathcal{P})$, and for all $i \geq 0$ $r_i \xrightarrow{\text{rp}} r_{i+1}$. The $\xrightarrow{\text{rp}}$ transition relation is defined over runtime programming states by the following operational semantics rules:

$$\frac{s \xrightarrow{P} s'}{(P, s, \hat{s}) \xrightarrow{\text{rp}} (P, s', \hat{s})} \ (20) \qquad \frac{\hat{s} \xrightarrow[P,s]{\mathcal{P}} \hat{s}'}{(P, s, \hat{s}) \xrightarrow{\text{rp}} (P, s, \hat{s}')} \ (21)$$

$$\frac{s \underset{P,\sigma/N,\hat{s}}{\overset{\mathcal{P}}{\Longrightarrow}} s' \quad \hat{s} \underset{P\,[\sigma/N],s'}{\overset{\mathcal{P}}{\longrightarrow}} \hat{s}'}{(P, s, \hat{s}) \xrightarrow{\text{rp}} (P\,[\sigma/N], s', \hat{s}')} \ (22)$$

Progress is thus expressed in a runtime programming trace in one of three ways: a transition of the running program (20), a transition of the patcher (21), or a synchronization between patcher and program, whereby the program is modified in accordance to a patch induced by the patcher (22). The execution of a patcher and a running program are interleaved, except for synchronized patch effect. This in line with the scheme of Fig. 6.

4.3 Patch compilation

We consider patch compilation comprises a set of assertions or actions `Aspects`, called compilation aspects, that relate to the notion of model-preserving patch as follows:

$$\forall \phi \in \texttt{Aspects}, P\,[\sigma/N] \models \phi \implies \sigma/N \in \texttt{patches}(P).$$

That is, if all compilation aspects are established for $P\,[\sigma/N]$, then σ/N is a model-preserving patch for P. An implication, rather than an equivalence above, acknowledges that patch compilation may be approximate, i.e., not recognize all model-preserving patches.

Per each $\phi \in \texttt{Aspects}$, an incremental compilation strategy is defined as a tuple

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

such that: φ_ϕ is a logical predicate over components called the incremental effort; $\mathcal{D}_\phi$ is a function of the form

$$\mathcal{D}_\phi : \texttt{Programs} \times \texttt{Components} \times \Sigma^* \to 2^{\texttt{Components}},$$

called the dependency context, such that given P, N and σ, $\mathcal{D}_\phi(P, N, \sigma)$ is a set of components in $P\,[\sigma/N]$, and may also include $O = P[\sigma]$; C_ϕ is a function with the same arguments as D_ϕ, called the compilation cost, that characterizes the time complexity for some algorithm that asserts φ_ϕ over $\mathcal{D}_\phi(P, N, \sigma)$, i.e., abusing notation as in Fig. 9, $C_\phi(P, N, \sigma) = O\left(\mathcal{D}_\phi(P, N, \sigma) \models \varphi_\phi\right)$; and ϕ, φ_ϕ and $\mathcal{D}_\phi$ are related by the following logical inference (again the same as in Fig. 9):

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

That is, if P is a program for which ϕ holds, and φ_ϕ holds for $\mathcal{D}_\phi(P, N, \sigma)$, then ϕ also holds for $P[\sigma/N]$. Note that we may have that $\mathcal{D}_\phi(P, N, \sigma) = \emptyset$, meaning no incremental compilation is required. This special case may occur when a patch preserves ϕ from P, or ϕ is implied by some other compilation aspects that operate in precedence in the context of compilation (e.g., as in Section 3 for refinement-level HTL components).

4.4 HTL instantiation

We provide next a description of the basic aspects of formal instantiation of runtime programming over HTL, in relation to the formal HTL syntax and semantics of [17].

Syntax. Every HTL component has an unique name at each level of scope, allowing for a trivial definition of component paths that conform to (1)–(3). The mapping of components to processes under constraints (4)–(5) holds by considering the processes of a component C to be $\texttt{processes}(C) = \bigcup_{m \in \texttt{Modes}^*(C)} \texttt{Tasks}(m)$, where $\texttt{Modes}^*(C)$ is the set of all modes defined recursively by C (including C if C is a mode), and $\texttt{Tasks}(m)$ is an unique set of tasks defined for a mode m, cf. [17].

Semantics. With regard to HTL semantics, the instantiation traits are as follows:
– The execution state of a component C, $(t, v, A) \in \texttt{states}(C)$ is defined by: $t \in \mathbb{Q}_{\geq 0}$, the execution time; a valuation v of task variables defined by C or parent components of C, plus communicators accessed by C; an an activation state A. The activation state A may be $\perp$ meaning the component is idle. Otherwise A is called an activation, and defined as follows: for a program P, A is a set of activations $\langle M, A_M \rangle$, one per module M in the program; for a module M, $A = \langle m, A_m \rangle$, where m is a mode in M, and A_m is an activation of m; for a mode m, $A = \langle a, A_R \rangle$, defined by $a \in \mathbb{Q}_0^+$, the start time of the current period of m, and an activation A_R of the refinement R of m, if one is defined, otherwise $A_R = \perp$.
– Initial states $s = (t, v, A) \in \texttt{init}(C)$ for a component C are all those with: a time t that is is harmonic with communicator periods accessed by C; an initialization of all task variables in C (cf. [17]); and s.t. every activation A_M of a module M within A has the form $A_M = \langle m_0, \cdot \rangle$, where m_0 is the start mode of module M.
– Quiescent states $(t, v, A) \in \texttt{qstates}(C)$ are defined if $A = \perp$ (C is idle); or, if $A \neq \perp$, then $s\lfloor C_0 \rfloor \in \texttt{qstates}(C_0)$ for all sub-components C_0 of C, and, additionally in case C is a mode m, $t = a + \Delta$ for $A = \langle a, \cdot \rangle$ where Δ is the period of m (t is a mode switching instant for m).
– In [17], transitions $\xrightarrow{C}$ for a component C are defined in line with component activations. Conditions (7)–(9) can be shown to hold, considering those component transitions and a notion of state projection as follows. We take $s\lfloor C_0 \rfloor = (t, v\lfloor C_0 \rfloor, A\lfloor C_0 \rfloor)$ for a sub-component C_0 of a component C and a state $s = (t, v, A)$ of C, where $A\lfloor C_0 \rfloor$ is the activation state of C_0 within A, and $v\lfloor C_0 \rfloor$ is a restriction of v for the variables of C_0.

Time determinism. Time-determinism is expressed over HTL traces as follows. Let $\texttt{Comms}(P)$ and $\texttt{Sensors}(P)$ to be the sets of all communicators and sensor communicators in a top-level program P, respectively. For two HTL traces $(t_i, v_i, A_i)_{i \geq 0}$ and $(t'_i, v'_i, A'_i)_{i \geq 0}$ of a time-deterministic program P, such that for every $c_s \in \texttt{Sensors}(P)$ we have $v_i(c_s) = v'_j(c_s)$ when $t_i = t'_j$ (the traces express the same timed sequence of inputs), it must be also that $v_i(c_o) = v'_j(c_o)$ for $c_o \in \texttt{Comms}(P) - \texttt{Sensors}(P)$ (the traces express the same timed sequence of outputs).

Runtime patches. All conditions for syntactic and semantic effect of a patch, (10)–(11) and (12)–(14) – are instantiated through the notions of component paths, initialization, quiescence, and state projection, and time-determinism defines the criterium for com-

pliance with correctness after patch effect. Thus, the set of model-preserving patches for a program P, $\texttt{patches}(P)$, is well defined.

Patch compilation. As discussed in Section 3.3, standard incremental HTL compilation subsumes all necessary aspects of patch compilation, except that of patch feasibility (15), equivalent to asserting eventual verification of (12)–(14) in the flow of a program P for a patch σ/N. Quiescence (12) will always eventually hold (as discussed in Section 3.2, top-level program patches are only considered through decomposition). So will (14), due to strict insulation of components $P - O$ from N in terms of activation state and task variables. The initialization requirement (13) will only hold if there is a guaranteed time of quiescence of O, that is also harmonic to the periods of all communicators accessed by N (in line with Fig. 12). This will hold by mere progress of time when we are adding N ($O = \bot$), or by well-formedness for refinement-level N (mode periods will not change from O to N). For top-level N replacing O, possible linear-time checks comprise a relation between $\mathtt{h}_N$ and h_O, taking h_X as the hyperperiod of communicators accessed by X. A sufficiency condition is that $h_O/h_N \in \mathbb{N}$, or, in inverse manner, that $h_N/h_O \in \mathbb{N}$, for the special case where O only contains modes of equal periods (as in the `MMode` patch of Fig. 13a).

5 Related work

Models for runtime software change have long been considered. Influential work can be found in [23, 8, 42, 16]. Even if these proposals have an high degree of generality, they still take into account elaborate notions of component interaction and specification, e.g., dependencies, communication, or specific traits for quiescence of components. This work draws inspiration from them, particularly [23] w.r.t. considering an abstract notion of quiescence, but defines comparatively simple abstract notions of component composition, initialization and quiescence.

The problem of verifying and integrating runtime software changes has led to a wide range of specialized compilation techniques, e.g., automatic derivation of patches from source code repositories [1], verified code generation [39, 6, 28], type safety inference for patches [29, 27, 2], or inference of "contextual side-effects" in concurrent programs [29, 30]. Our interest was to put forward a framework such that these techniques can in principle be characterized in incremental form. We find that other principled abstractions can also be important for scalable runtime programming, such as proof-carrying code for runtime certified compilation [31], or modular frameworks for component-based systems [3, 41].

There is active interest in forms of runtime patching for real-time systems at large e.g., see [36, 40, 13, 33, 34]. Earlier influential work can be found in [37], with regard to assurances of dependability, and in [38], for a characterization of component design and timing issues in component re-configuration [38]. Mode change protocols [35] are also highly relevant, as discussed for HTL runtime patching, by providing a formulation to reason on aspects such as schedulability for a runtime switch in a real-time system. Asserting schedulability of a real-time program can be an unscalable process, as in HTL. Paths for scalable schedulability analysis include incremental verification [12], runtime certification through schedule-carrying code [18], and temporal isolation schemes [9].

In [22], some of the ideas in this paper and HTL runtime patching were discussed in preliminary short form, and considering only the particular context of real-time systems and HTL. The general formulation here is far more elaborate, not restricted to real-time systems, and HTL is strictly a case-study instantiation.

6 Conclusion

We proposed runtime programming, a methodology for flexible software design defined by recurrent runtime patches to a program. The presentation comprised a formulation of concepts, its corresponding formalization, and a case-study instantiation for the HTL language. The core concepts were that of model-preserving runtime patches, and an incremental patch compilation methodology for scalability. We also reasoned this foundation could be used to handle a wider context of runtime patches, by the notion of patch decomposition. The HTL case-study demonstrated applicability of all these aspects.

References

1. Arnold, J., and Kaashoek, M. Ksplice: Automatic rebootless kernel updates. In *EuroSys* (2009).
2. Bierman, G., Parkinson, M., and Noble, J. UpgradeJ: Incremental Typechecking for Class Upgrades. In *ECOOP* (2008).
3. Bozga, M., Sfyrla, V., and Sifakis, J. Modelling synchronous systems in BIP. In *EMSOFT* (2009).
4. Brewer, E. Lessons from Giant-Scale Services. *IEEE Internet Computing* (2001).
5. Brinkschulte, U., Schneider, E., and Picioroaga, F. Dynamic real-time reconfiguration in distributed systems: timing issues and solutions. In *ISORC* (2005).
6. Cai, H., Shao, Z., and Vaynberg, A. Certified Self-Modifying Code. In *PLDI* (2007).
7. Clarke, E., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, 1999.
8. Cook, J., and Dage, J. Highly reliable upgrading of components. In *ICSE* (1999).
9. Craciunas, S., Kirsch, C., Payer, H., Röck, H., and Sokolova, A. Programmable Temporal Isolation through Variable-Bandwidth Servers. In *IEEE SIES* (2009).
10. Curino, C., Moon, H., Ham, M., and Zaniolo, C. The PRISM Workwench: database schema evolution without tears. In *IEEE ICDE* (2009).
11. Dumitraş, T., and Narasimhan, P. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise systems. In *ACM/IFIP/USENIX Middleware* (2009).
12. Easwaran, A., Shin, I., Sokolsky, O., and Lee, I. Incremental schedulability analysis of hierarchical real-time components. In *EMSOFT* (2006).
13. Estevez-Ayres, I., Garcia-Valls, M., Telematica, D., Almeida, L., Aveiro, P., and Basanta-Val, P. Solutions for Supporting Composition of Service-Based Real-Time Applications. In *ISORC* (2008).
14. Ghosal, A., Iercan, D., Kirsch, C., Henzinger, T., and Sangiovanni-Vincentelli, A. Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code. In *APGES* (2007).
15. Ghosal, A., Kirsch, C., Henzinger, T., Iercan, D., and Sangiovanni-Vincentelli, A. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT* (2006).
16. Gupta, D., Jalote, P., and Barua, G. A formal framework for on-line software version change. *IEEE TSE* (1996).

17. Henzinger, T., Kirsch, C., Marques, E., and Sokolova, A. Distributed, Modular HTL. In *IEEE RTSS* (2009).
18. Henzinger, T., Kirsch, C., and Matic, S. Schedule-carrying code. In *EMSOFT* (2003).
19. Three-tank system videos. http://htl.cs.uni-salzburg.at/examples.html.
20. Iercan, D. *Contributions to the Development of Real-Time Programming Techniques and Technologies*. PhD thesis, Politehnica University of Timisoara, 2008.
21. Internal Electrotechnical Commission. *IEC 61499: Function blocks for industrial-process measurement and control systems*, 2005.
22. Kirsch, C., Lopes, L., and Marques, E. Semantics-Preserving, Incremental Runtime Patching of Real-Time Programs. In *APRES* (2008).
23. Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE TSE* (1990).
24. Kramer, J., and Magee, J. Self-managed systems: an architectural challenge. In *IEEE FSE* (2007).
25. Love, J., Jariyasunant, J., Pereira, E., Zennaro, M., Hedrick, K., Kirsch, C., and Sengupta, R. CSL: A Language to Specify and Re-specify Mobile Sensor Network Behaviors. In *IEEE RTAS* (2009).
26. Lublinerman, R., Szegedy, C., and Tripakis, S. Modular Code Generation from Synchronous Block Diagrams — Modularity vs. Code Size. In *POPL* (2009).
27. Martins, F., and Lopes, L. Towards Safe Programming of Wireless Sensor Networks. *Elsevier EPTCS* (2010).
28. Myreen, M. Verified Just-In-Time Compiler on x86. In *POPL* (2007).
29. Neamtiu, I., and Hicks, M. Safe and timely updates to multi-threaded programs. In *PLDI* (2009).
30. Neamtiu, I., Hicks, M., Foster, J., and Pratikakis, P. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL* (2008).
31. Necula, G. Proof-carrying code. In *POPL* (1997).
32. OSGi Alliance. *OSGi Service Platform Core Specification, Version 4, Release 4.1*, 2007.
33. Pfeffer, M., and Ungerer, T. Dynamic real-time reconfiguration on a multithreaded Java-microcontroller. In *ISORC* (2004).
34. Rasche, A., and Polze, A. Dynamic reconfiguration of component-based real-time software. In *WORDS* (2005).
35. Real, J., and Crespo, A. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems* (2004).
36. Richardson, T., Wellings, A., Dianes, J., and Díaz, M. Providing Temporal Isolation in the OSGi Framework. In *JTRES* (2009).
37. Sha, L. Dependable System Upgrade. In *RTSS* (1998).
38. Stewart, D., Volpe, R., and Khosla, P. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE TSE* (1997).
39. Stoyle, G., Hicks, M., Bierman, G., Sewell, P., and Neamtiu, I. Mutatis mutandis: safe and predictable dynamic software updating. *POPL* (2005).
40. Thramboulidis, K., and Zoupas, A. Real-time Java in control and automation: a model driven development approach. In *ETFA* (2005).
41. Tripakis, S., Lickly, B., Henzinger, T., and Lee, E. On relational interfaces. In *EMSOFT* (2009).
42. Wermelinger, M. Towards a chemical model for software architecture reconfiguration. In *CDS* (1998).