Runtime Programming through Model-Preserving, Scalable Runtime Patches*

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

[1] Department of Computer Sciences, University of Salzburg, {*ck,anas*}@*cs.uni-salzburg.at*

[2] CRACS & INESC-Porto LA, Faculdade de Ciências, Universidade do Porto, {*lblopes,edrdo*}@*dcc.fc.up.pt*

***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 and scalability. Model preservation means that a runtime patch preserves the programming model in place for programs — in terms of syntax, semantics, and correctness properties — as opposed to an "ad-hoc", disruptive operation, or one that requires an extra level of abstraction. Scalability means that, for practicality and performance, the effort in program compilation required by a runtime patch should ideally scale in proportion to the change induced by it. We formulate runtime programming over an abstract model for component-based concurrent programs, defined by a modular relation between the syntax and semantics of programs, plus built-in notions of initialization and quiescence. The notion of a runtime patch is defined over these assumptions, as a model-preserving transition between two programs and respective states. Additionally, we propose an incremental compilation framework for scalability in patch compilation. The formulation is put in perspective through a case-study instantiation over a language for distributed hard real-time systems, the Hierarchical Timing Language (HTL).**

I. 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. Related work in many diverse research communities (e.g.: programming languages [1]; operating systems [2]; databases [3]; large-scale web servers [4]; real-time control systems [5]; or sensor networks [6], [7]) and in industry (e.g., the OMG OSGi [8] and IEC 61499 [9] standards) typically tends to take a partial and domain-specific view of the problem. 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 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 (e.g., see [10]): the evolving program is the "plant", the patcher is the "controller", and runtime patches define the "control".

*This work has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23), an Elise Richter Fellowship (Austrian Science Fund V00125), and Fundação para a Ciência e Tecnologia funds PTDC/EIA/71462/2006 and SFRH/BD/29461/2006.

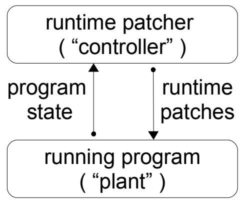


Fig. 1. Runtime programming.

More concretely, we consider the runtime programming of component-based concurrent programs, and present methodologies for its design and validation. The proposal is structured around two core sustaining principles, model preservation and scalability, first introduced in short form in [11], as follows.

Model preservation. By model preservation we mean that a runtime patch should 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 of model preservation 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.

For model preservation, a runtime patch must define a switch between two programs and their states, and one that ensures correctness of operation. A runtime patch is formulated as the replacement of a (syntactic) component within a program, that affects the behaviour of concurrent processes described by the program (semantics) in instantaneous manner. The assumption is an 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, that define conditions for graceful component activation and deactivation, respectively. We require that: processes of the replaced component terminate in a quiescent state; processes of the new component start from a valid state; and that processes of other components are unaffected by the patch effect.

These requirements define a base provision for a correct and continuous flow between the two programs that run before and after patch effect, but may not be sufficient, however, to comply with particular correctness properties at stake. The problem is that the operation of programs started in a live manner through a patch, can differ from their standard operation from scratch (overall initial conditions), and deviate from correctness. Addressing this may require verification, taking into account the difference of state space for patch-induced operation of programs. All these aspects may render a given patch infeasible, or too complex to analyze, but in that case the runtime patching operation may work as a base inductive case for decomposing non model-preserving patches into sequences of model-preserving patches.

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, conducted in an online or partially offline context, such as checking a model-preserving nature for patches, and other aspects like code generation or re-linking. 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 portion of the program affected by the patch.

With scalability in mind, we propose a patch compilation framework, that defines how to conduct patch compilation incrementally, and inherently characterizes its scalability. The base idea is that for each aspect of compilation of a patch over a given program, an incremental effort is conducted considering only a dependency context of components in the program. The portion of dependency context and the complexity of the incremental effort characterize scalability of patch compilation in the space and time dimensions. For instance, if the dependency context is the full program, or the incremental effort has intractable complexity, then compilation is not scalable. The framework is a generalization of the base ideas of a modular compilation framework already defined in [12], and proposed first informally for runtime patching in [13], for the context of component-based systems and runtime programming.

Contribution and structure. Our contribution is a characterization of runtime programming in the terms above. It comprises three parts: the definition of program model assumptions (Section III), that serve as base for runtime programming; the definition of runtime programming over that component-based program model, with runtime patching at its core (Section IV); and a characterization of incremental patch compilation (Section V). Additionally, we define runtime programming in formal terms (Section VI). The formalization addresses matters of detail and preciseness that can however otherwise be skipped for understanding the general formulation.

Throughout the text, we put the formulation in perspective with a case-study instantiation, taking in context a component-based language for real-time distributed control systems, the Hierarchical Timing Language (HTL) [12], [14], which is briefly introduced first (Section II). The case study makes the point that runtime programming can concretely provide flexibility to computer systems, given a reasonable degree of compositionality in program specification and scalability in program compilation. It also illustrates some of the difficulties when compositionality and scalability are missing. The details of the HTL instantiation in formal terms are not provided in this paper, but can be found in our technical report [15].

The paper ends with a complementary discussion of related work (Section VII), and some concluding remarks (Section VIII).

II. Case study

HTL is a component-based coordination language for distributed real-time systems [12], [14]. An HTL program is a hierarchical tree-like structure composed of other components called modules and modes, which in turn may define inner programs through a relation of hierarchical refinement. These components map to the execution of real-time tasks on a distributed platform, with Logical Execution Time guarantees [16].

As a running HTL example, we consider the real-world application of a three-tank system (3TS), described in [17]. The 3TS, with structure depicted in Fig. 2, 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. The control runs on three hosts: two of the hosts have direct access to the pumps, and the remaining host is used as a monitoring interface to an operator.

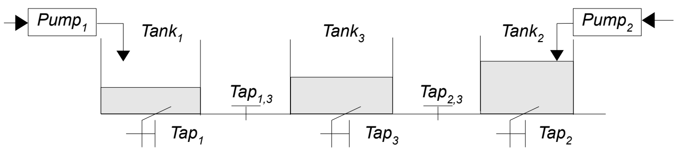


Fig. 2. The three-tank system (3TS) [17].

An HTL program for the 3TS is shown in Fig. 3. The program is presented in detail in [17], and some videos demonstrating it at work can be found at [18]. We consider an adaptation of the original program, by letting the P controllers mentioned above run at 1 Hz, and the PI controllers run at 2 Hz, rather than a fixed frequency of 2 Hz for all control in the original program [17]. In Fig. 3, the syntactic structure of the adapted program (left) and a possible execution of it (right) are shown.

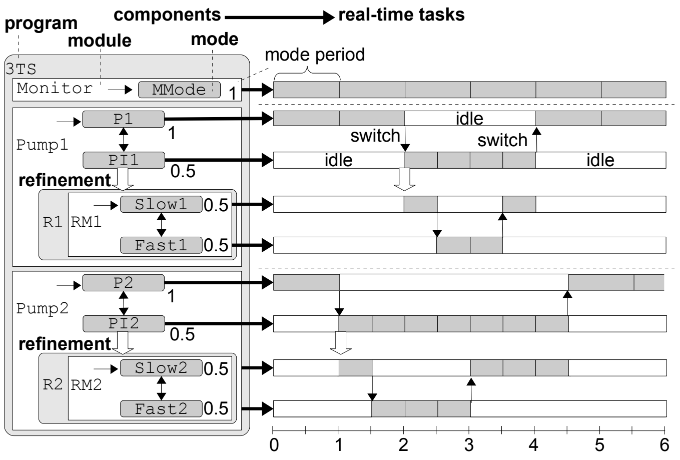


Fig. 3. An HTL program for the 3TS (adaptation from [17]).

An HTL program defines a set of module components that execute concurrently, which can be distributed module-wise across different hosts in a network. In Fig. 3, the `3TS` program consists of three concurrently running modules `Monitor`, `Pump1`, and `Pump2`, each mapped to one of the 3TS hosts.

A module is defined by a set of mode components, with one mode identified as start mode, e.g., `P1` for module `Pump1` in Fig. 3, and some mode switching logic expressed by conditions over communicator variables. A module executes by activating one mode component at a time, beginning with the start mode, and evaluates mode switching logic to determine the next mode to execute in sequence. For instance, in Fig. 3, 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 a set of real-time tasks, and their invocation over a fixed time period, called the mode's period, e.g., mode periods are 1 or 0.5 in Fig. 3. 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. 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, e.g., schedulability of computation, and in so, leveraging the effort of HTL compilation [12], [14], as we illustrate later in the text. Refinement programs are active when their parent mode is also active and in time-synchronized form at mode switching instants. In Fig. 3, each PI-control mode (`PI1`, `PI2`) in the 3TS program is refined by a program (`R1`, `R2`) with a single module (`RM1`, `RM2`) that defines the "slow"-PI and "fast"-PI control modes required for 3TS control. For the sample execution shown, mode `PI1` and its refinement program `R1` are active in the interval $(2, 4)$, and, likewise, `PI2` and `R2` are active in the interval $(1, 4.5)$.

III. Program model

We take as assumption a programming model for concurrent, 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 concurrent programs that allows for incremental modifications at runtime with a well-defined functional effect.

Syntax and semantics. We consider a program specified by composition of (syntactic) components whose execution is defined by sets of (semantic) concurrent processes. This is shown in Fig. 4 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.

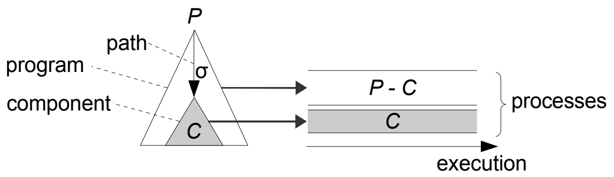


Fig. 4. Program model assumptions — syntax and semantics.

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. 4, 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. 5 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. 5: 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.

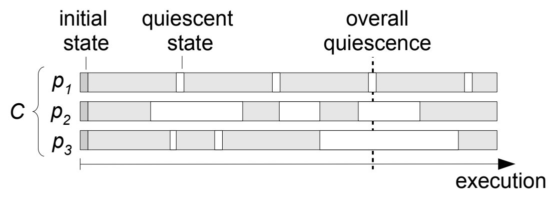


Fig. 5. Program model assumptions — initialization and quiescence.

The fundamental traits we put forward, clean formulations of concurrency, initialization, and quiescence, are necessary to reason on non-disruptive runtime programming. Concurrency allows to reason on piecewise changes to a program, and initialization and quiescence define when and how it is appropriate to do so. Instantiation of these traits can be found in related work on runtime modifications to a system, e.g., detecting inactive kernel functions in operating systems [2], reaching annotated program points in multi-threaded programs [1], or dynamic update models for distributed systems [19].

The HTL case. Let us analyze how the abstract programming model assumptions instantiate for HTL.

The syntax and semantics of HTL are modularly related. Each component describes an isolated set of processes in the form of real-time tasks, as illustrated in the 3TS example of Fig. 3. Each component is also uniquely named at each scoping level [12], [14] (e.g., all modules in the same program), hence component paths are naturally defined using component names, e.g., the path of mode `P1` in Fig. 3 can be expressed as `3TS.Pump1.P1`.

HTL component quiescence can be seen as expressed by intervals of idleness, when a component's execution has no side effects, or atomic instants of mode switching evaluation in all modes of a component. For example, in Fig. 3, `PI1` and all its sub-components are idle in intervals $(0, 2)$ and $(4, 6)$, and mode switching is evaluated every 0.5 seconds in interval $(2, 4)$. Each individual mode, module, or refinement level programs is guaranteed to reach quiescence, all of them and their sub-components eventually synchronize in mode-switching terms. Regarding top-level programs, this might not be true, except for certain sub-classes of programs, e.g., those in which all modes in a top-level module have equal periods, allowing for synchronization at least on a hyperperiod basis. For instance, in Fig. 3, if no mode switching occurs in both `Pump1` and `Pump2` after time 4.5, the whole program will not quiesce indefinitely, as mode switching will always subsequently be evaluated at different times in the two modules.

A notion of component initialization is also finally in place. It is required that each module, when initialized, begins execution from its start mode, as mentioned previously. Other initialization constraints must be observed that are specific to the component interaction model in HTL. Component interaction is done through special variables called communicators. A communicator is a global, top-level program variable that has an associated period for access by real-time tasks in a mode. Fig. 6 depicts a possible communicator

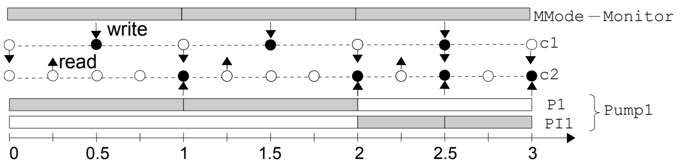


Fig. 6. HTL communicator interaction.

interaction between the modules `Monitor` and `Pump1` of the 3TS program, using communicators `c1` and `c2` with periods 0.5 and 0.25, respectively. The tasks of modes `P1` and `PI1` in module `Pump1` write to `c2`, and read from `c1`. Tasks in mode `MMode` within module `Monitor` read `c2`, and write to `c1`. A communicator can only be logically read or written at times that are multiples of its period, as illustrated in the figure. We stress the term logical time [16]: the behavior of the program must be as if communicator reads and writes (or other events like task completions) occur at specified logical instants, despite the fact that the associated platform events may happen at different, varying times [12], [14]. The initialization constraint w.r.t. communicator access is that that any period instance of a mode must start at a time that is multiple of all communicator periods it accesses.

IV. Formulation of runtime programming

We define runtime programming over the program 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.

Runtime patch. A runtime patch is illustrated in Fig. 7. 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 $P\,[\sigma/N]$.

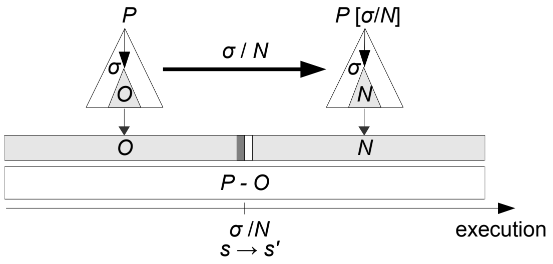


Fig. 7. A runtime patch.

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.

In Fig. 8, 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 $P\,[\sigma/N]$.

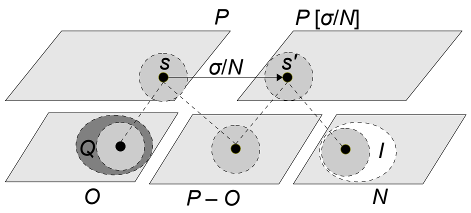


Fig. 8. Patch effect and the state space of components.

Runtime patcher. A runtime patcher 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. We abstract away from the problem of how the patches are derived and the associated complexity, (e.g., as in the controller synthesis problem [10], [20]) and focus on the verification of a given patch through compilation (as in the controller verification problem). We do not also consider the actual design of the runtime patcher and supporting system, which can have elaborate requirements, e.g., in the vein of reconfigurable "live systems" [21], [22].

The notions of runtime patcher and runtime patching define the possible executions of a runtime programming system, in the form illustrated in Fig. 9. The figure shows that, starting from an initial configuration where program P_0 is active, a patcher $\mathcal{P}$ has the ability of inducing patches $\sigma_0/N_0, \sigma_1/N_1, \ldots$ over program execution. The resulting program sequence is

$$P_0, P_1 = P_0\,[\sigma_0/N_0], P_2 = P_1\,[\sigma_1/N_1],\ \ldots,$$

and the resulting program state sequence is

$$s_0,\ \ldots\ , s'_0, s_1,\ \ldots\ , s'_1, s_2,\ \ldots\ , s'_2,\ \ldots,$$

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

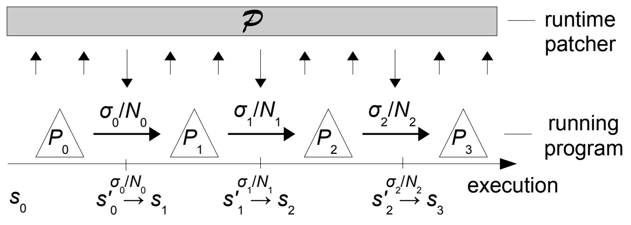
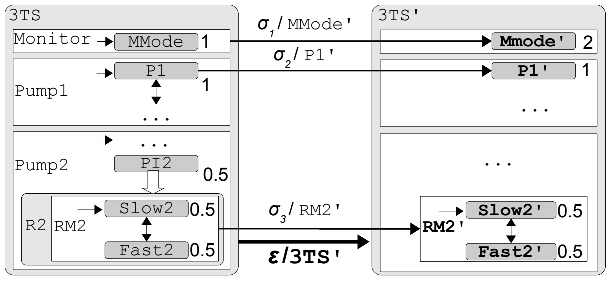
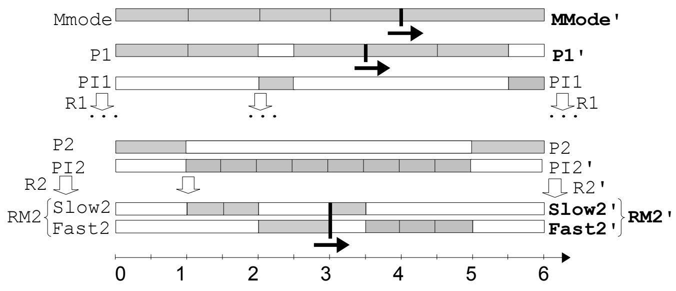


Fig. 9. Runtime patcher and program execution.

HTL patching example. We consider some example patches over the HTL 3TS example, depicted in Fig. 10. The figure shows the syntactic changes defined by the patches (Fig. 10a), and their possible semantic effect (Fig. 10b), assuming they can be applied in any order from time 3. The patches can be understood as individually applied in self-contained manner, or as part of a larger patch that proceeds in decomposed form, a concept we describe later in the text. For illustrative purposes, one can consider that the runtime patcher is some piece of software interfacing with a distributed HTL runtime system [14], [23] that is extended for runtime code instrumentation.



(a) Syntactic change.



(b) Semantic effect.

Fig. 10. 3TS program patch.

The example patches of Fig. 10 syntactically change `3TS` by replacing top-level modes `MMode` (σ_1/`MMode'`) and `P1` (σ_2/`P1'`), plus refinement module `RM2` (σ_3/`RM2'`). The program timing is maintained, except for a change of period between `MMode` and `MMode'`. It is assumed that `MMode'` has an initialization constraint, such that it can only start at time instants multiples of 2, due to some communicator access definitions. From time 3, when the patches become available, we have that: the patch over `RM2` can proceed at time 3 immediately, since it reaches a convenient quiescent (mode switch evaluation) state; the patch over `P1` is delayed until time 3.5 for quiescence (at time 3, `P1` is active); and the patch over `MMode` is delayed until time 4, due to the initialization constraint on `MMode'`, even though time 3 is suitable in quiescence terms.

In general, an HTL runtime patch can define the replacement of modes, modules, and refinement programs, or the addition or removal of modules. The addition or removal of modes and refinement programs would disrupt the specification of their parent component, hence are not well-formed patches. Individual patches affecting different top-level modules in a program do not generally guarantee eventual quiescence (synchronized evaluation of mode switching), as mentioned earlier. But, at the top-level program specification, it could be considered also that the communicator set of the program is changed. We do not consider this case, without loss of expressiveness: a patch that alters the communicator set can be shown equivalent to a patch between programs with the same communicator set.

Model preservation. Model preservation, as stated in the introduction, 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. This might not hold by the definition of patch effect alone. A patch defines an instantaneous switch between two programs, thus its effect can be explained alone by the programming model in place. Its effect is also safe, as it provides a continuous flow between programs (proper quiescence, initialization, and isolation). However, correctness may be compromised for the operation of the program that is started through patch effect.

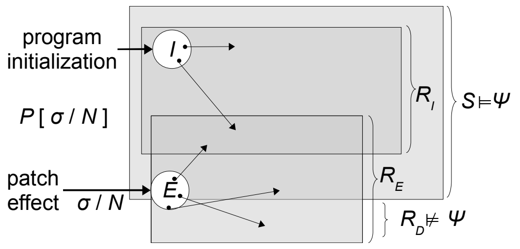


Fig. 11. Deviation from correctness after patch effect.

The problem is illustrated in Fig. 11. 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. 11, where ψ does not hold. Thus, as in the supervision of control systems [10], we wish that "undesirable events" are avoided. From a compilation perspective, the problem can be addressed by essentially taking into account the difference in state-space between execution from scratch, and execution induced by a patch.

This view of model preservation does not necessarily require a strict relation between patch-induced traces and standard program traces, though they may of course relate in some manner. Other approaches are possible. For instance, convergence of patch-induced traces to standard program operation, with possible deviant behavior over a transient period, has been considered in [24]. The later is only appropriate as far as transient incorrect behavior is acceptable. In earlier work [13], we discussed a view of model preservation where the effect of a patch to a program should be strictly equivalent to another program expressing the same functionality switch, but this is too rigid.

Note also that there might be properties of correctness that are difficult or impossible to analyze beforehand through verification. Redundancy and fault isolation methodologies have been proposed, e.g. [5], [25], [26], that counter for deviation of correctness after runtime modifications to a program. We do not consider this type of methodologies, but their use is not ruled out by our component-based program model assumptions, e.g., an analytic redundancy relation between components [5] can be instantiated in our framework.

HTL and model preservation. The key aspect of program correctness in HTL is defined by a property called time determinism. Time determinism relates to the values of communicators (Fig. 6) over time as follows. A program is time-deterministic if for every timed sequence of inputs, corresponding to the values over time of a subset of communicators called sensor communicators, the program always yields a unique timed sequence of outputs, given by the values of all other communicators over time. A time-deterministic program can guarantee predictable and portable functionality over any given platform with sufficient computational resources [12], [14]. The property is derived by verification of race-free communication interaction plus schedulability of computation and network transmissions, as we describe later in the text.

Potential deviation from time determinism could result from (unverified) patch effect. The combination of active concurrent tasks in different modes may be different from an execution from scratch, in the sense that the combination would not be reachable from initial conditions. Time determinism can be compromised by patch effect, e.g., if there is no feasible real-time schedule for the resulting workload. In the 3TS example this is not an issue, since the control over each of the 3TS pumps is functionally independent, i.e., a combination of simultaneous P and slow/fast PI-control modes is possible in any case.

Patching scope and decomposition. Recall the possibly delicate requirements posed on runtime patching: synchronization of quiescence, valid initialization, isolation of effect, and compliance with correctness after patch effect. As a result, large patches may be infeasible, for instance because they replace components that quiesce in unsynchronized manner, or that are too complex to verify w.r.t. correctness after patch effect.

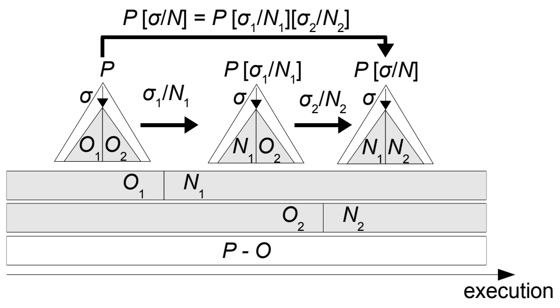


Fig. 12. Patch decomposition.

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. 12. 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. 12, 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, as in mode change protocols for real-time systems [27], 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. 12, where $\perp$ stands for an undefined component. The downtime may be necessary for several reasons, for instance time consuming state-transfer (e.g., [28], [29]), or synchrony of effect with other patches in context.

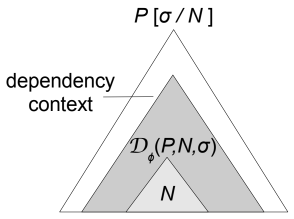
HTL and patch decomposition. The example of Fig. 10 can be seen as an HTL patch $\varepsilon/\texttt{3TS}'$, that is decomposed in the three model-preserving patches discussed earlier, $\sigma_1/\texttt{MMode}'$, $\sigma_2/\texttt{P1}'$, and $\sigma_3/\texttt{RM2}'$, taking effect in interval $(3, 4)$. The decomposition can be called asynchronous in the sense of mode-change protocols for real-time systems [27], as it defines a mix of old and new functionality over the transient period of effect, e.g., $\texttt{RM2}'$ executes together with `P1` until time 3.5 and with `MMode` until time 4. A synchronous decomposition [27] would be possible for the same patch, by conveniently removing components that are to be patched, first, inducing some downtime at respective paths, to achieve a synchronous effect of all patches, later. In [15], we give such an example for the same overall patch.

Synchronous and asynchronous decompositions represent a trade-off between several factors such as verification effort, promptness from request to completion, and downtime of affected components [27]. An asynchronous decomposition requires extra effort to verify correct operation over the transient period when old and new tasks mix, but can have faster effect and induces no downtime. A synchronous decomposition is less complex to verify, at the expense of slower effect and downtime.

V. Patch compilation

Patch compilation is the process of verifying and integrating a runtime patch in a runtime programming system. Taking the controller-plant analogy of Fig. 1 in context again, compilation relates to controller verification, though not to controller synthesis [10]. Patches are "given" in this sense, though the problem of "synthesizing" them could be formulated for the runtime patcher. Verification of a runtime patch over a given program needs to establish the conditions for model-preservation formulated previously. For this, 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. Compilation may also include the process of deriving a patch decomposition, in which case patches in a decomposition must also be individually compiled. Other aspects such as code generation or relinking should also be dealt with.

Several specialized techniques may apply in this context, see [1], [2], [6], [30]–[34]. 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 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 [12], and generalize it for our abstraction of component-based software.



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

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

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

Fig. 13. Incremental patch compilation.

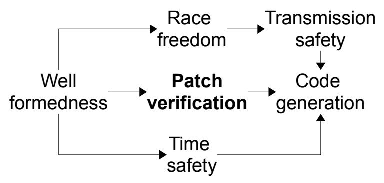
The proposal is illustrated in Fig. 13. The idea is that patch compilation should consist of an incremental effort operating over the dependency context of components related to a patch. Per each compilation aspect ϕ, say for instance code generation, the dependency context of a patch σ/N over P, $\mathcal{D}_\phi(P,N,\sigma)$, shown left in the figure, identifies the portion of $P\ [\sigma/N]$ 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 $\mathcal{D}_\phi(P,N,\sigma)$, 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 $\mathcal{C}_\phi(P,N,\sigma) = \mathcal{O}\left(\mathcal{D}_\phi(P,N,\sigma) \models \varphi_\phi\right)$, 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 $I(\phi) = (\varphi_\phi, \mathcal{D}_\phi, \mathcal{C}_\phi)$ 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 [23], [35]–[37]. By our formulation in Fig. 13, $P \models \phi \wedge \mathcal{D}_\phi(P, N, \sigma) \models \varphi_\phi$ is a sufficiency criteria for $P\,[\sigma/N] \models \phi$. The dependency context and compilation cost depend on the choice of φ_ϕ, which represents the balance between precision and effort. Note that we are not characterizing the complexity of compilation, but present a principled approach for dealing with it. A complexity analysis can be done for actual instantiation, as we discuss for HTL next.

HTL patch compilation. Standard compilation of HTL programs comprises several aspects, and precedences between them, in the manner shown in Fig. 14a. For patch compilation, one additional aspect of patch verification is also needed and shown in the figure. An HTL compiler takes into account a program's specification, a mapping of top-level modules to platform hosts, and a characterization of worst-case execution times for task computation and worst-case transmission times for communicator value transmission in the platform's real-time network. A program needs to be checked first for well formedness, i.e., compliance with syntactic constraints. These include trivial syntactic verifications, as for any language, but also particular hierarchical refinement and Logical Execution Time constraints [12], of core importance in HTL. Next, the compiler checks time determinism. This is addressed by verifying three sub-properties: race freedom — the absence of races in communicator writes; time safety — schedulability of computation per each host; and transmission safety — schedulability of network broadcasts for communicator value propagation. A race-free, time-safe, and transmission-safe program is time-deterministic [12]. At the end of the compilation process, E-code [23] is generated.

This compilation process has been characterized in modular, incremental terms [12]. Incremental compilation strategies were defined for the problem of compiling component N that changes over program P, yielding $P\,[\sigma/N]$, or equivalently in the context of patching, compiling a patch σ/N over P. These are summarized in the table of Fig. 14b, in terms of the dependency context $\mathcal{D}_\phi(P, N, \sigma)$ and complexity class of compilation cost $\mathcal{C}_\phi(P, N, \sigma)$, introduced previously. Checking well-formedness and code generation need only take N as dependency context and requires linear time in the size of the syntactic representation of N. If N is a refinement component, like in the $\sigma_3/\mathtt{RM2}'$ patch of Fig. 10, refinement constraints alone guarantee that race freedom, time safety, and transmission safety are preserved from P, so these need not be checked. Otherwise, e.g. $\sigma_1/\mathtt{MMode}'$ in Fig. 10, they need to take the entire program $P\,[\sigma/N]$ in context, but can be performed in linear time, with the exception of time safety. The latter, even if it can be done in isolation for the host where N will run, may induce an exponential time effort.

Overall, the scalability of HTL patch compilation may thus be compromised by the verification of time safety over top-level components. We have that the complexity scales exponentially with the number of top-level modules per host [12], even if the refinement-level part of a program need not need be accounted for, and time safety can be proceed modularly per host. In the 3TS patch example of Fig. 10, each of the patches affects functionality running on a different host, and each host runs only a single module which is verifiable with pseudo-polynomial complexity (depends on the numerical values of mode periods) [12]. However, if all modules in the 3TS system were to run in the same host, that would not be the case. The type of decomposition could then play a key role, by minimizing as much as possible the effort in verifying time safety, as discussed above in connection to mode-change protocols [27], and in more detail in [15]. For the base problem, a number of established techniques could improve scalability, e.g., incremental scheduling analysis [38], schedule-carrying code [39], or temporal isolation schemes [40]. Finally, a runtime programming implementation can conceivably deal with time safety analysis offline, so that the supporting runtime system is not affected in an unscalable manner.



(a) Compilation aspects.

ϕ	$\mathcal{D}_\phi(P, N, \sigma)$		$\mathcal{C}_\phi(P, N, \sigma)$	
	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 [12]).

For patch compilation, additional compilation work must be done. In line with the formulation of Section IV, 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 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. Note that potential deviation from correctness, i.e. 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 [12]. 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. Hence time-determinism can be ensured in all possible executions after patch effect. Regarding code generation, the E-code format of [23] should be tuned to counter for runtime relinking, but can otherwise maintain the same modular structure, and be incrementally generated.

Thus, patch verification merely requires checking patch feasibility, i.e., ensure the initialization, quiescence, and isolation requirements of runtime patching eventually hold in the execution of a program P for semantic effect of a given patch σ/N. 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 [12], [15]. The initialization requirement holds if $O = P\,[\sigma]$ always quiesces at instants that are consistent startup times for top-level N, as defined by communicator accesses, and illustrated in Fig. 6. This is ensured by well-formedness of N alone (mode periods will not change from O to N). Otherwise, for top-level N, simple checks over the set of communicators accessed by O and N can be done in linear time, as indicated in Fig. 14b for the patch verification aspect. The details are given in our technical report [15].

VI. Formalization

In this section we provide a definition of runtime programming in formal terms. It addresses matters of detail and preciseness in regard to the characterization of runtime programming in the previous sections, but otherwise expresses the same concepts. We proceed by characterizing the program model assumptions, the formulation of runtime programming, and incremental patch compilation, in this order. The HTL instantiation of this formalization can be found in [15].

A. Program model assumptions

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

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

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. 4. 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

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

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. 4.

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 [41] 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.

B. 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 $\bot$.

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 conditions express 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 $N = \bot$, a component addition if $O = \bot$, and a component replacement otherwise ($O, N \neq \bot$).

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:

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

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

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

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,

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

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.,

$$\forall \psi \in \Psi,\ \psi(s_0, s_1, s_2, \ldots) \tag{15}$$

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. 12):

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

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

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

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

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 ($N_1 = \sigma/\perp$ over P, $N_2 = \sigma/N$ over $P\,[\sigma/\perp]$). Note that conditions (16), (18), and (19) imply a finite recursion over (17).

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

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

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 $s \overset{\mathcal{P}}{\underset{P,\sigma/N,\hat{s}}{\Longrightarrow}} s'$ 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 $i \geq 0$ $r_i \xrightarrow{\texttt{rp}} r_{i+1}$. The $\xrightarrow{\texttt{rp}}$ transition relation is defined over runtime programming states by the following operational semantics rules:

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

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

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. 9.

Thus, a runtime programming trace has an associated sequence of programs $P_0, P_1 = P_0\ [\sigma_0/N_0], P_2 = P_1\ [\sigma_1/N_1]$, $\ldots$, and a corresponding program state sequence $s_0, \ldots, s'_0, s_1, \ldots, s'_1, s_2, \ldots, s'_2, \ldots$, such that $s_i, \ldots, s'_i$ are traces (for $i = 0$) or patch-induced traces (for $i > 0$) of P_i, and $s'_i \xrightarrow{\sigma_i/N_i} s_{i+1}, i \geq 0$. The definition of runtime patch effect, comprising conditions (12)–(14) implies by construction that proper quiescence, initialization, and isolation of patch effect, are observed when $s'_i \xrightarrow{\sigma_i/N_i} s_{i+1}, i \geq 0$. Additionally, the model-preserving condition (15) ensures correctness of operation for patch-induced traces $s_i, \ldots, s'_i, \forall i > 0$.

C. Patch compilation

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

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

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 $I(\phi) = (\varphi_\phi, \mathcal{D}_\phi, \mathcal{C}_\phi)$, such that: φ_ϕ is a logical predicate over components called the incremental effort; $\mathcal{D}_\phi$ is a mapping $\mathcal{D}_\phi : \texttt{Programs} \times \texttt{Components} \times \Sigma^* \rightarrow 2^{\texttt{Components}}$, 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 $O = P[\sigma]$; $\mathcal{C}_\phi$ 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. 13, $\mathcal{C}_\phi(P, N, \sigma) = \mathcal{O}\,(\mathcal{D}_\phi(P, N, \sigma) \models \varphi_\phi)$; and ϕ, φ_ϕ and $\mathcal{D}_\phi$ are related by the following logical inference (again the same as in Fig. 13):

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

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 V for refinement-level HTL components).

VII. Related work

This paper builds on prior work. In [13], some of the ideas in this paper were discussed in preliminary form, considering only the particular context of real-time systems and HTL. The formulation here is more general, for abstract component-based systems, and HTL is strictly a case-study instantiation of the general framework. In [12], runtime patching was mentioned as a possible application of incremental HTL compilation, but the necessary extensions were not discussed. Our technical report [15] provides complementary detail on some aspects of this paper, and the extended abstract in [11] summarizes its overall proposal in short form.

Models for runtime software change have long been considered. Influential work can be found in [19], [22], [24], [26]. Even if these proposals have a high degree of generality, they still take into account elaborate notions of program interaction and specification, e.g., dependencies, communication, or specific traits for quiescence of components. We consider comparatively simpler abstract notions of component composition, initialization, and quiescence. Our point was to capture an abstract view of compositionality in component-based programs and formulate runtime programming over it.

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 [2], verified code generation [30]–[32], type safety inference for patches [1], [6], [34], or inference of "contextual side-effects" in concurrent programs [1], [33]. Our interest was to put forward a framework such that these techniques can in principle be characterized in incremental, scalable form. We find that other principled abstractions can also be important for scalable runtime programming, such as proof-carrying code for runtime certified compilation [42], or modular frameworks for component-based systems [35], [36].

There is active interest in forms of runtime patching for real-time systems, e.g. [43]–[47]. Earlier influential work can be found in [5], with regard to assurances of dependability, and in [29], for a characterization of component design and timing issues in component re-configuration [29]. Mode change protocols [27] are also highly relevant by providing a formulation to reason on aspects such as schedulability for a runtime switch in a real-time system. As discussed for HTL, asserting schedulability of a real-time program can be an unscalable process, but principled methodologies can be used to overcome the problem [38]–[40]. Code generation may also not scale and modular generation schemes should be used in this regard, e.g., as in [23] (for HTL) or [37].

VIII. 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 two core contributions were the concept of model-preserving runtime patch, and a framework for incremental patch compilation. We believe these are principled approaches to reason on runtime software change in non-disruptive manner, and characterize its scalability. The HTL case-study illustrated the possibilities and difficulties of the runtime programming methodology, with the overall conclusion that our methodology can indeed be used for flexible software change at runtime, for reasonable degrees of compositionality in program specification and scalability in program compilation.

Ackowledgements. We wish to thank all the anonymous reviewers for their comments on previous versions of this paper.

References

[1] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in Proc. PLDI. ACM, 2009.
[2] J. Arnold and M. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in Proc. EuroSys. ACM, 2009.
[3] C. Curino, H. Moon, M. Ham, and C. Zaniolo, "The PRISM Workwench: database schema evolution without tears," in Proc. ICDE. IEEE, 2009.
[4] E. Brewer, "Lessons from Giant-Scale Services," IEEE Internet Computing, 2001.
[5] L. Sha, "Dependable System Upgrade," in Proc. RTSS. IEEE, 1998.
[6] F. Martins and L. Lopes, "Towards Safe Programming of Wireless Sensor Networks," Elsevier EPTCS, 2010.
[7] J. Love, J. Jariyasunant, E. Pereira, M. Zennaro, K. Hedrick, C. Kirsch, and R. Sengupta, "CSL: A Language to Specify and Re-specify Mobile Sensor Network Behaviors," in Proc. RTAS. IEEE, 2009.
[8] OSGi Service Platform Core Specification, Version 4, Release 4.1, http://www.osgi.org, OSGi Alliance, 2007.
[9] IEC 61499: Function blocks for industrial-process measurement and control systems, http://www.iec.ch, Internal Electrotechnical Commission, 2005.
[10] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," SIAM J. Control Optim., 1987.
[11] C. Kirsch, L. Lopes, E. Marques, and A. Sokolova, "Runtime Programming through Model-Preserving, Scalable Runtime Patches," in Proc. FACS, Doctoral Track. Springer-Verlag, 2010, 4-page extended abstract.
[12] T. Henzinger, C. Kirsch, E. Marques, and A. Sokolova, "Distributed, Modular HTL," in Proc. RTSS. IEEE, 2009.
[13] C. Kirsch, L. Lopes, and E. Marques, "Semantics-Preserving, Incremental Runtime Patching of Real-Time Programs," in Online Proc. APRES, 2008.
[14] A. Ghosal, C. Kirsch, T. Henzinger, D. Iercan, and A. Sangiovanni-Vincentelli, "A hierarchical coordination language for interacting real-time tasks," in Proc. EMSOFT. ACM, 2006.
[15] C. Kirsch, L. Lopes, E. Marques, and A. Sokolova, "Runtime Programming through Model-Preserving, Scalable Runtime Patches," Department of Computer Sciences, University of Salzburg, Tech. Rep. 2010-08, 2010.
[16] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A Time-triggered Language for Embedded Programming," Proc. IEEE, 2003.
[17] D. Iercan, "Contributions to the development of real-time programming techniques and technologies," Ph.D. dissertation, Politehnica University of Timisoara, 2008.
[18] "Three-tank system videos," http://htl.cs.uni-salzburg.at/examples.html.
[19] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," IEEE TSE, 1990.
[20] S. Jiang and R. Kumar, "Supervisory control of discrete event systems with CTL* temporal logic specifications," SIAM J. Control Optim., 2006.
[21] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in IEEE FSE, 2007.
[22] M. Wermelinger, "Towards a chemical model for software architecture reconfiguration," in Proc. CDS. IEEE, 1998.
[23] 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," LNCS Science of Computer Programming, 2010.
[24] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," IEEE TSE, 1996.
[25] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise systems," in Proc. ACM/IFIP/USENIX Middleware, 2009.
[26] J. Cook and J. Dage, "Highly reliable upgrading of components," in Proc. ICSE. ACM/IEEE, 1999.
[27] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," Real-Time Systems, 2004.
[28] U. Brinkschulte, E. Schneider, and F. Picioroaga, "Dynamic real-time reconfiguration in distributed systems: timing issues and solutions," in Proc. ISORC. IEEE, 2005.
[29] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," IEEE TSE, 1997.
[30] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: safe and predictable dynamic software updating," Proc. POPL, 2005.
[31] H. Cai, Z. Shao, and A. Vaynberg, "Certified Self-Modifying Code," in Proc. PLDI. ACM, 2007.
[32] M. Myreen, "Verified Just-In-Time Compiler on x86," in Proc. POPL. ACM, 2007.
[33] I. Neamtiu, M. Hicks, J. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming," in Proc. POPL. ACM, 2008.
[34] G. Bierman, M. Parkinson, and J. Noble, "UpgradeJ: Incremental Typechecking for Class Upgrades," in Proc. ECOOP. LNCS, 2008.
[35] M. Bozga, V. Sfyrla, and J. Sifakis, "Modelling synchronous systems in BIP," in Proc. EMSOFT. ACM, 2009.
[36] S. Tripakis, B. Lickly, T. Henzinger, and E. Lee, "On relational interfaces," in Proc. EMSOFT. ACM, 2009.
[37] R. Lublinerman, C. Szegedy, and S. Tripakis, "Modular Code Generation from Synchronous Block Diagrams — Modularity vs. Code Size," in Proc. POPL. ACM, 2009.
[38] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee, "Incremental schedulability analysis of hierarchical real-time components," in Proc. EMSOFT. ACM, 2006.
[39] T. Henzinger, C. Kirsch, and S. Matic, "Schedule-carrying code," in Proc. EMSOFT. ACM, 2003.
[40] S. Craciunas, C. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Programmable Temporal Isolation through Variable-Bandwidth Servers," in Proc. SIES. ACM, 2009.
[41] E. Clarke, O. Grumberg, and D. Peled, Model Checking. MIT Press, 1999.
[42] G. Necula, "Proof-carrying code," in Proc. POPL. ACM, 1997.
[43] T. Richardson, A. Wellings, J. Dianes, and M. Díaz, "Providing Temporal Isolation in the OSGi Framework," in Proc. JTRES. ACM, 2009.
[44] K. Thramboulidis and A. Zoupas, "Real-time Java in control and automation: a model driven development approach," in Proc. ETFA. IEEE, 2005.
[45] I. Estevez-Ayres, M. Garcıa-Valls, D. Telematica, L. Almeida, P. Aveiro, and P. Basanta-Val, "Solutions for Supporting Composition of Service-Based Real-Time Applications," in Proc. ISORC. IEEE, 2008.
[46] M. Pfeffer and T. Ungerer, "Dynamic real-time reconfiguration on a multithreaded Java-microcontroller," in Proc. ISORC. IEEE, 2004.
[47] A. Rasche and A. Polze, "Dynamic reconfiguration of component-based real-time software," in Proc. WORDS. IEEE, 2005.