ABSTRACT

Strong pressure on deployment of embedded control systems on a low-cost hardware leads to the need of optimizing software architectures to minimize resource demands. Nevertheless, releasing the resources not needed in specific phases of system execution is only rarely supported by today’s component frameworks, mainly since information about the system state is spread over several components, which makes the idea hard to implement.

The paper introduces a formal model of property networks allowing for efficient capture of modifications of architecture-relevant information and shows, how this model can be used to employ the concept of modes for system architectures in hierarchically nested component systems.

Categories and Subject Descriptors
F.1.2 [Modes of Computation]: Interactive and reactive computation; C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms
Design

Keywords
embedded-systems, real-time systems, software variability, modes for software architectures, change propagation, SOFA

1. INTRODUCTION

Embedded control systems have become an inherent part of various devices used in industry and academia, ranging from spacecraft, avionic, automotive, and medical applications to home automation. Such typically single-purpose systems are often deployed on specialized hardware, which imposes numerous constraints. In addition to being required to be predictable, the system behavior has to meet time- and resource-related constraints.

Motivated by lowering the cost, there is strong pressure on deploying and running control systems on limited hardware. This implies optimizations of software architecture for minimal resource demands, including releasing resources in the phases of execution when the resources are not needed. This leads to requiring variability of system architecture.

However, the forms of system variability [3] known, e.g., from desktop systems, are too loose for embedded control systems, since in these systems it is essential to predict resource requirements. At the same time, the restricted but well-defined forms of architecture variability known as modes for software architectures [11] are hard to implement in hierarchically nested component systems. This is mainly because the information about system state is spread among several components; consequently the architecture modifications implemented by hand are potentially error prone, or even unfeasible due to the complexity of dependencies among components.

1.1 Motivation Example

Different variants of software architecture are needed in various kinds of control systems, as they, for example, calibrate sensors in the initial phase, perform specific phases of control loops, and clean up in a final phase. As an example consider a control system designed to drive a mobile robot equipped with two sensors (a front bumper and a camera) together with two actuators (left and right wheel engines).

Robot’s control logic implements exploitation of the environment composed of areas and enclosed by a wall — black circles correspond to towns, gray lines to roads and the light-gray background represents the rest of the environment (Figure 1). The robot is seeking for a town or road (for “civilization”). Once the robot finds one, it discovers towns by following roads until it gets acquainted with the whole environment; then the robot stops and switches to stand by mode.

The underling control software operates in six variants of its architecture: (i) civilization search (CSV), (ii) on-road movement (RMV), (iii) town discovery (TDV), (iv) initialization (INIT) and, additionally, (v) stand by (SB).

Each variant requires a different set of system resources (e.g., sensors and wheel drivers). Obviously, their allocation...
and release is to be handled adequately during the architecture variant switch.

An overview of the architecture variants is shown in Figure 2, where variants are introduced as different combinations of software components. Each of the components is labeled to indicate the architecture variant in which it is employed (e.g., Bumper driver is needed in the INIT and CSV variant).

Figure 2: Key software components used in the mobile robot control software (labeled by the variants of architecture in which they are employed)

In principle, the code realizing an architecture variant switch can be either handwritten or (preferably) generated in an automated way. Even this simple example demonstrates that changing architectural variants by handwritten code (considering resource handling) would be hard to implement and consequently error prone. Hence, the rest of the paper focuses on the second option.

1.2 Mode Change Request and Deadline

The paper presents a theoretical model that allows for employing the concept of software architectural modes [11].

The key concepts related to software architecture modes include:

A mode change request is an event that causes a transition from a current (“old”) operating mode to a new one (realized by a mode change mechanism). Typically, a mode change request cannot be accepted when another mode transition is in progress ([23]).

The time necessary to calculate the next mode has to be predictable and short enough to meet a mode change deadline introduced in order to prevent overwriting of available resources.

1.3 Problem Statement

Stemming from [24, 10, 23], the requirements on a mode change mechanism can be summarized as follows.

(R1) Preservation of the Assume-guarantee Principle. This stresses the need of explicitly stating under which condition on the environment a component guarantees its correct functionality.

(R2) Composability of the Mode Change Mechanism. The mode change mechanism has to be distributed in components and its effect must be composable along with component hierarchy.

(R3) Not Polluting the Existing Interface and Business Logic. Embedding of mode change mechanism into a component should not modify the existing component interfaces and its business logic code.

(R4) Feasible and Predictable Mode Change Mechanism Time and Space Complexity. The mode change mechanism has to guarantee meeting the mode change deadlines and be feasible in terms of space requirements.

To our best knowledge, the existing modular and component-based development frameworks targeting the domain of real-time systems (overviewed e.g., in [12]) do not provide explicit mechanism meeting all the requirements R1–R4. In this respect, MyCCM-HI [25] belongs to the most advanced frameworks, but it still fails in fully addressing R3 and R4.

1.4 Goals and Structure of the paper

The first objective of the paper is to introduce a theoretical model supporting design of a mode change mechanism satisfying all the requirements R1–R4. Being proposed for real-time hierarchical component systems, it introduces the concepts of component properties and property networks allowing exchange of architecture-relevant information among components. In addition, these concepts allow constructing an oracle inferring, in constant time, the architecture consequences of a mode change throughout all the levels of a component hierarchy.

Another objective is to analyze the model in terms of time and space complexity.

The final goal is to report on a proof-of-concept implementation applied to a non-trivial case-study.

To reflect the goals, the paper is structured as follows:

Section 2 provides an informal overview of the approach taken in this paper while Section 3 introduces the new theoretical base for mode change propagation modeling. Section 4 describes how this theoretical base can be employed in designing a mode change mechanism meeting the requirements R1 – R4. Section 5 and 6 discuss a proof-of-concept implementation and a non-trivial case-study; further, Section 7 contains an evaluation of the proposed method in-
null
3. THEORETICAL BASE

This section introduces theoretical background of property networks - novel formalism with intuitive semantic designed to be suitable for mode change propagation.

3.1 Properties and a Property Network

In this section we assume a property set $\mathcal{P} = \{p_1, \ldots, p_m\}$ where $m \geq 1$ is finite. Each property $p_i$ is associated with its valuation $\alpha_i$ from a finite domain $D(p_i)$.

**Definition 3.1 (property function).** Let $\mathcal{P}$ be a set of properties. A function $f$ operating on a vector of valuations $[\alpha_1 \ldots \alpha_m]$ of a vector of properties $[p_1 \ldots p_m]$, $p_i \in \mathcal{P}$, $p_i \neq p_j$, $1 \leq i, j \leq m$, yielding a valuation of another property $p_i$, i.e., $f : D(p_1) \times \cdots \times D(p_m) \rightarrow D(p)$ is called a property function.

**Definition 3.2 (input and output property set).** Assume a set of properties $\mathcal{P}$ and a property function $f$ mapping the valuations of $[p_1, \ldots, p_m]$ to a valuation of $p$. The set $\{p_1, \ldots, p_m\} \subseteq \mathcal{P}$, the input property set of $f$ is denoted $\mathcal{I}(f)$; and $p \in \mathcal{P}$, the output property of $f$, is denoted $\mathcal{O}(f)$.

**Definition 3.3 (property network).** Let $\mathcal{P}$ be a set of properties and $\mathcal{F}$ a set of property functions upon $\mathcal{P}$. The property network $\mathcal{N}$ is a triple $\mathcal{N} = (\mathcal{P}, \mathcal{F}, \mathcal{L})$ where $\mathcal{L}$ represents the linking of $\mathcal{I}(f)$ and $\mathcal{O}(f)$ for all $f \in \mathcal{F}$.

The concept of linking is introduced as follows: $\mathcal{N}$ can be represented as a bipartite graph $G(N)$, where one type of nodes (places) represents the properties and the other type of nodes (transitions) represents functions (Figure 3). And the directed arcs of the graph reflect $\mathcal{L}$ as follows: An arc leads from a property $p$ (place $p$) to a function $f$ (transition $f$) if $p \in \mathcal{I}(f)$; in a similar vein, an arc leads from $f$ (transition $f$) to a property $p$ (place $p$) if $p = \mathcal{O}(f)$.

3.2 Network Reaction Semantics

**Definition 3.4 (network state, triggered property).** A Network state $S = (V, T)$ of a network $\mathcal{N} = (\mathcal{P}, \mathcal{F}, \mathcal{L})$ is the pair of valuation $V$ of the properties in $\mathcal{P}$ and the set $T \subseteq \mathcal{P}$ of triggered properties.

**Definition 3.5 (e-network, initial network state).** Let us assume a property network $\mathcal{N} = (\mathcal{P}, \mathcal{F}, \mathcal{L})$. An executable network, e-network, corresponding to $\mathcal{N}$ is the tuple $\mathcal{E} = (\mathcal{P}, \mathcal{F}, \mathcal{L}, I)$ where $I = (V^0, T^0)$ is the initial network state.

**Definition 3.6 (e-network execution semantics).** Execution semantics of an e-network $\mathcal{E} = (\mathcal{P}, \mathcal{F}, \mathcal{L}, I)$, $I = (V^0, T^0)$ is defined as a transition relation $R$ on network states. Assuming a state $S_k = (V^k, T^k)$, $k \geq 0$, the transition from $S_k$ to $S^{k+1}$ is in $R$ if $S^{k+1}$ can be derived from $S^k$ in the following way: Starting with $V^{k+1} = V^k$ and $T^{k+1} = \emptyset$, each function $f \in \mathcal{F}$ such that $i(f) \cap T^k \neq \emptyset$ is applied on the properties in $i(f)$ (the valuation of which is determined by $V^k$), yielding in consequence a modification of the valuation of $p \in o(f)$ in $V^{k+1}$ and the membership of $p$ in $T^{k+1}$.

Figure 5: Property networks: a) finite run, triggering has a limited impact when assigning to $p_2$, b) infinite run with a stable state despite circular relationship of properties (detected in the state denoted $\checkmark$) c) finite run without a stable state (detected in the state denoted $\times$)

Case a is related to acyclic networks, an example is shown in Figure 5 a).

In the more interesting case b), stabilized states are repeated with the period $j - k$ (while the valuation of "intermediate" states is not modified) as it is illustrated in Figure 5 b) where the state 2 is the first stabilized state ($k = 2$) and the state 4 is the following stabilized state ($j = 4$). When observing the run, it is sure in state 4, that state 2 is stable,
because all the states between 2 and 4 have the same valuation of all properties and the same properties are triggered in states 2 and 4. Note that the stable state is derived from the first stabilized by setting its set of triggering indices to $\emptyset$, in the example the stable state is $(V^2, \emptyset)$ and it was derived from state 2 by setting $T^2 = \emptyset$.

Contrary to this, in Figure 5 c) a stable state will be never reached, because states will be repeated with period 4 as could be first observed in the network run in state 5, which is the same as state 1, but valuations in states between 1 and 5 differ.

Assuming an e-network $E = (P, F, L, I)$, a property $p \in P$ can be in $o(f)$ of more than one function $f \in F$. Because the order in which effect of function evaluation is applied is not defined (according to Definition 3.6), there can exist more than one run of an e-network $E$ as shown in Figure 6, where value of $p_k$ can be set as an effect of $f_3$ or $f_4$. Thus, it is not possible to define the stable state of $E$ unambiguously, but such a definition is possible for the following class of e-networks:

$$F = \{ f_1, f_2, f_3, f_4, f_5, f_6, f_7 \}$$

$$F' = (F \setminus \{ f_3, f_4 \}) \cup \{ f_1 \}$$

$$= (F \setminus \{ f_3, f_4, f_5, f_6 \}) \cup \{ f_1 \}$$

$$= (f_1, f_2, f_3, f_4, f_5, f_6)$$

Figure 7: Illustration of Lemma 3.12 and Theorem 3.13 - replacing of several functions modifying the same output property by a single function.

Proof (basic idea): Lemma 3.12 can be systematically applied to the properties breaking definition 3.9; in this process no new properties breaking 3.9 can appear.

Definition 3.14 (alternating network). A well-defined e-network $E = (P, F, L, I)$ is called alternating if its run does not contain a stable state.

Note, that in the run of an alternating network, the valuation of properties is periodically modified, so that there is no stabilized state (according to 3.8 b) even if it was $T^{k+\ell} = T^j + 1$, it would be $V_k \neq V^k+i$). An example is shown in Figure 5 c) where states 1 and 5 are identical but valuations of $p_1$ and $p_2$ in states between 1 and 5 differ.

Assume a well-defined e-network $E = (P, F, L, I)$, where $I = (V^0, T^0)$, $T^0 = \{ p_i \}$ ($T^0$ contains a single property $p_i$). Algorithm react calculates the stable state of $E$:

**Theorem 3.15.** Assume a well-defined e-network $E = (P, F, L, I)$, where $I = (V^0, T^0)$, where $T^0$ contains a single triggered property $p_i$. The algorithm 1 determines correctly the stable state of the $E$ run in a finite number of steps.

Proof (basic idea): The algorithm follows straightforwardly the network semantics (Definition 3.6). Because each function is only once in the set TriggeredFunctions and each property $p$ can be modified at most by one function ($E$ is well-defined), the value $V^0[p]$ is always unambiguous and thus the order in which the functions are executed is irrelevant.

### 3.3 Oracle Construction

The aim of oracle is to infer in constant time architecture consequences of a mode change throughout all the component hierarchy. It is constructed as a finite-state automaton composed of stable network states with transitions representing consequences of propagation of all the possible triggering events. In principle, each transition represents a run of the network. An example of an oracle is in Figure 8.

Construction of the oracle is based on two algorithms (i) the algorithm react simulating network propagation of a triggering event in the network (ii) the oracle algorithm exploring all the reachable stable states.
Algorithm 1 react(\(\mathcal{E} = (P, F, \mathcal{L}, I)\) with \(I = (V^0, \{p_i\})\))

create sets \(\text{TriggeredFunctions, EvaluatedFunctions}\);
for all \(f \in F: p_i \in i(f)\) do
insert \(f\) into \(\text{TriggeredFunctions}\)
end for
set \(V = V^0\)
while \(\text{TriggeredFunctions} \neq \emptyset\) do
\(\text{EvaluatedFunctions} \leftarrow \text{TriggeredFunctions}\)
set \(\text{TriggeredFunctions} \leftarrow \emptyset\)
while \(\text{EvaluatedFunctions} \neq \emptyset\) do
\(f \leftarrow \text{RemoveOne}(\text{EvaluatedFunctions})\)
\(V'[\sigma(f)] \leftarrow f(V[p_1], \ldots, V[p_k])\)
where \(p_1, \ldots, p_k \in i(f)\)
for all \(g \in F: p_i \in i(g)\) do
insert \(g\) into \(\text{TriggeredFunctions}\)
end for
end while
if \((V', \text{TriggeredFunctions})\) is stabilized state then
return \((V', \emptyset)\)
else if \((V', \text{TriggeredFunctions})\) is alternating then
throw alternatingNetworkException
end if
set \(V = V'\)
end while
return \((V', \emptyset)\)

As was already discussed in Section 2.1, time complexity of react algorithm evaluation is likely to be beyond acceptance threshold in various control systems. This section describes construction of an oracle that can return in constant time a stable state by pre-computing the stable states of all e-networks runs that have to be considered in the application.

Definition 3.16 (oracle, triggering event). An oracle is a state machine, where a state \(s_j\) represents the stable state of the well-defined e-network \(\mathcal{E}_j = (P, F, \mathcal{L}, I_j)\), \(I_j = (V^0_j, \emptyset)\).
A transition \(s_k \xrightarrow{(p_i, \alpha_i)} s_{k+1}\) represents the run of the e-network \(\mathcal{E}_k = (P, F, \mathcal{L}, I'_k)\), where \(I'_k = (V^0_k, \{p_i\})\), \(V^0_k\) is obtained from \(V_0\) by setting \(p_i\) to \(\alpha_i\).
We say that the e-network \(\mathcal{E}_k\) was transformed to \(\mathcal{E}_k'\) by the triggering event \(e_i = (p_i, \alpha_i)\).

Consider the e-network \(\mathcal{E} = (P, F, \mathcal{L}, I^0)\) and a set of triggering events \(E\). The following algorithm \(\text{PSM}\) (Property State Machine) constructs a state machine (oracle) representing transitive closure of stable states of all e-networks \(\mathcal{E}'\) which can be transitively transformed from \(\mathcal{E}\) by applying the triggering events from \(E\).
Algorithm 2 constructs the state space of a transitive closure in the breadth-first style as shown in Figure 8. Its optimized version reducing the number of calls of react (Algorithm 1) is available in [21].

4. MODE CONCEPT FORMALIZATION

In the domain of control systems it is crucial to articulate exact semantic of all involved concepts and provide complexity assessment of all related algorithms. The following section outlines formalization of the concepts employed. Some definitions and theorem proofs are stated informally, or even

Algorithm 2 PSM(\(\mathcal{E} = (P, F, \mathcal{L}, I), E\))

create queue \(Q\)
create sets \(\text{States, Transitions}\)
enqueue \(\text{initialState}\) into \(Q\)
while \(Q \neq \emptyset\) do
\(\text{dequeue state} = (V, \emptyset)\) from \(Q\)
for all \(e_i = (p_i, \alpha_i) \in E\) do
\(\text{NewState} \leftarrow \text{react}((P, F, \mathcal{L}, (V[p_i] \leftarrow \alpha_j, \{p_j\})))\)
if \(\text{NewState} \notin \text{States}\) then
add \(\text{NewState}\) into \(\text{States}\)
add \((\text{state, NewState}, e_i)\) into \(\text{Transitions}\)
add \(\text{NewState}\) into \(Q\)
end if
end for
end while
return \((\text{States, Transitions})\)

omitted, due to the space limit (their full-fledged version is available in [21]).

4.1 Mode

In the following text, we consider primitive components containing only executive code (not any other component), and composite components containing other components.
In general, a component can exist in several variants of its internal architecture (subcomponents, bindings and component attributes) – its modes.

Definition 4.1 (component mode). Assume a software component \(C\); its mode \(M\) is the triple \((S^M_C, B^M_C, R^M_C)\) where
- \(S^M_C\), subcomponent set is the set of subcomponents present in \(M\) in the architecture of \(C\) at the first level of nesting (subcomponent frames).
- \(B^M_C\), binding set is the set of the bindings present in \(M\) in the architecture of \(C\) at the first level of nesting.
- \(R^M_C\) denotes the component attribute set of \(C\) in \(M\).

Definition 4.2 (component mode set). Consider the component \(C\). The set of all its modes is called component mode set (denoted \(M_C\)).
### 4.2 Component And Application Properties

**Definition 4.3** (mode condition). Assume a component $C$ with $M_C = \{M^1_C, \ldots, M^n_C\}$ and a property set $P_C = \{p_1, \ldots, p_m\}$. The mode condition $\Gamma_C$ is a function on $P_C$ yielding an $M_C \in M_C$, i.e., $\Gamma_C : D(P_C) \rightarrow M_C$.

**Definition 4.4** (application property set). Assume an application $A$ containing a set of components $C_A = C_1, C_2, \ldots, C_k$; each of them being associated with its property set $P_{C_i}$. The set $P_A = \bigcup_i (P_{C_i})$ is called the application property set of $A$.

It is advantageous to represent component mode of a component $C_i$ as one of the properties $P_{C_i}$ to capture mode changes as applications of property functions. The key related concept is well-defined $P_A$.

**Definition 4.5** (well-defined application property set, function $find$). Application property set $P_A$ is well-defined if in $A$ for each component $C_i$ associated with $P_{C_i}$ and $M_{C_i}$, it holds, that $P_{C_i}$ contains a property $p_{C_i, \text{mode}}$ with the domain $D(p_{C_i, \text{mode}}) = M_{C_i}$. Let us assume existence of a function $f$:

$$f : C_A \rightarrow P_A$$

returning a mode property $p_{C_i, \text{mode}}$ for any component $C_i$ in $A$.

**Definition 4.6** (well-defined application property network).

Application property network $N = (P, F, L)$ is well-defined, if $\forall f \in F$ it holds, that all the properties in $\text{in}(f) \cup \text{out}(f)$ are associated with components on the same or neighboring levels of nesting.

### 4.3 Inferring an Architecture Variant

Since the component architecture is formed by a component hierarchy where each component has its mode (Definition 4.1), the current application architecture variant can be defined as follows:

**Definition 4.7** (Current architecture variant). The set of components $C_i$ and component modes $M_i$ in $S = \{(C_1, M_1), (C_2, M_2), \ldots, (C_k, M_k)\}$ is called *current architecture variant* if

- $S$ contains $(C_T, M_T)$ for top-level component $C_T$.
- $\forall (C_i, M_i) \in S$, $S$ contains $(C_k, M_k)$, where $C_k \in S^M_C$, and does not contain any other pair.

Since a mode of a component determines its current subcomponents, bindings and attributes, comprehensive information about the current architecture variant can be obtained by a top-down inspection of component modes for given $P_A$ and $V_A$ are known.

The following algorithm $\text{getArch}$ (Algorithm 3) determines modes of all components present in application architecture in a particular mode. Input of the algorithm is a valuation $V_A$ of properties in $P_A$ and a top level component $C_{top}$. Output of the algorithm is current architecture variant in terms of Definition 4.7.

**Theorem 4.8.** Algorithm 3 is correct and its time complexity is linear in the number of all components in $A$.

Proof: directly from algorithm description. ■

### 5. CASE STUDY REVISED

In this section, the case study introduced in Section 1.1 is analyzed in terms of the presented concept - architecture variants shown in Figure 1 are elaborated via component modes. Specifically, a detailed architecture, a key part of the property network and the corresponding oracle (PSM) are presented and illustrated in Figure 9. As an example, the (INIT) architecture variant (Section 1) is in terms of Definition 4.7 reflected as INIT = \{cApplication, mInitApp\}, (eBrain, mInitBrain), (cInitBrain, false), \ldots\,\}, the civilization search variant (CSV) is reflected as CSV = \{cApplication, mExploration\}, (cBrain, mWordSearch), (cSearchStrategy, mCivSearch), \ldots\,\}.

The properties characterizing the application are listed together with their domains in the top-center of Fig. 9, e.g., property $m_{\text{modeBrain}}$ can have the valuation $m_{\text{InitBrain}}$, $m_{\text{WordSearch}}$, or $m_{\text{StandBy}}$. In a similar vein, triggering events are listed in the right-top of Fig. 9, e.g., $e_7$ sets the value of the property $\text{done}_{\text{InitBrain}}$ to true.

Assuming the left-most state of the PSM as the initial property state, $e_7$ causes transition of property valuation to the next stable state as indicated in the figure. The key difference between these two states is that valuation of the property $m_{\text{modeBrain}}$ is changed from $m_{\text{InitBrain}}$ to $m_{\text{WordSearch}}$ and this causes a change in the application variant obtained using Algorithm 3 ($\text{getArch}$).

PSM was obtained by applying Algorithm 2 (PSM) which uses also Algorithm 1 (react). The e-network serving as input to the PSM algorithm is also partially depicted in Fig. 9, e.g., value of the property $m_{\text{modeBrain}}$ is set by the function $f_{\text{modemodeBrain}}$ taking $\text{wordSearchDone}$, $\text{initDone}$ as arguments.

The function determining valuation of the property $m_{\text{SearchStrategy}}$ is more complex: taking three arguments ($\text{townDiscDone}$, $\text{groundType}$ and $m_{\text{SearchStrategy}}$), it is an example of a function where $\text{out}(f) \in \text{in}(f)$.

The alternative b) in definition of a stabilized state (Definition 3.8) is illustrated by functions taking as arguments properties $\text{done}_{\text{townDisc}}$ and $\text{townDiscDone}$ leading to an infinite run.

Overall, the initial e-network contains 13 properties and thus (at most) 13 functions while the PSM has 24 stable states. For the sake of lucidity, only interesting fragments are depicted in Figure 9.

### 6. PROOF OF THE CONCEPT

This section present a proof-of-concept implementation.
Figure 9: Component architecture of the robots’ control software with a property network and a part of the corresponding PSM. (The mode properties of primitive components are not shown, since they can be easily derived from the mode properties of their parent components.)

built on the hierarchical component system SOFA-HI [22] following the model presented in Sections 3 and 4.

6.1 Overall Strategy

In principle, it is necessary to (i) represent properties and property values, (ii) pre-compute the oracle (PSM, Algorithm 2) and (iii) represent PSM at run-time and finally (iv) implement variant switching (employing Algorithm 3 and representation of PSM).

Basically, (i), (iii) (iv) have to be a part of the run-time platform, while (ii) can be advantageously a part of deployment tools. Each of (i), (iii) (iv) was at run-time represented by a generated system component.

6.2 Challenging Issues

The proof-of-concept implementation had to address the following three challenging issues:

A) Achieving architecture variant change atomicity

The reaction to a triggering event has to be completed, before processing of any other triggering event starts [23], atomicity of triggering event processing has to be guaranteed. In the implementation this was achieved by introducing an event queue buffering the incoming triggering events if necessary.

B) Global validity of property identification

Because the property identifiers have to be unique in the context of the whole application, they have to be generated which makes them potentially hard-to-read. To address the issue, in the implementation, the component API is extended by dedicated local methods generated by a deployment tool.

The methods convert local identifiers to the those globally generated, so the component API contain functions like e.g., setModeProperty(propertyLocalId, value).

C) Property function specification

In order to compose property networks, a dedicated specification language has to be introduced. Key constructs include property declarations, their valuation via constants, function declarations, equality, simple relational and boolean operators (for expressing mode change conditions).

As an implementation decision if the function is not defined on a subset of input values, the user is warned (and for the time being the function does not modify the output value).

A fragments of specification in the simple language used in the proof-of-concept implementation is shown in boxes in Figure 9, e.g., mode worsStrategySearch ≠ ”townDisc” ⇒ townDiscDone = false

7. EVALUATION

7.1 Property Design Experience

Subjectively, the semantics of the property network
(change in one component influences others as described by a set of property functions) is easy-to-comprehend. Basically the design of property functions and the property network is intuitive and straightforward. Tools checking whether e-networks are well-defined and featuring stable states proved invaluable in the development process.

To introduce property network mechanism in the existing component-based system SOFA-HI a feasible effort was required (approximately 2 man months). Since SOFA-HI implementation was built with the help of model-based technologies a great part of the e-network supporting tools could be generated by enhancing the meta-models. A dedicated API was provided to allow for an easy way of triggering event from component code (Section 6.2 B).

7.2 Space and Time Complexity

At Deployment Time the react algorithm (1) is evaluated. Its time complexity is inherently exponential in the number of application properties. Finiteness was already shown by proving the Theorem 3.15.

Time complexity of the PSM algorithm (2) is proportional to $|S| \cdot |E_A| \cdot T_{react}$, where $|S|$ is the number of stable states, $|E_A|$ stands for the number of possible triggering events. $T_{react}$ is the time complexity of the react algorithm. Even if it is possible to optimize the number of $T_{react}$ invocations ([21]), the complexity of PSM algorithm remains exponential in the number of properties.

On the other hand, since the PSM algorithm is not executed at run-time (it is executed only at deployment phase), its exponential complexity does not cause a major issue considering a real-life number of properties.

At run-time PSM prepared at deployment time is applied to determine architecture variant switches.

The number of PSM states can be in the worst case exponential in the number of properties – the upper bound is $(\max(|D(p_i)|))^{|P_A|}$. However, the state space is pruned by the relations implied by property functions causing that a reaction to an event change values of several properties simultaneously. The other factor pruning the state space is that relatively small domains are considered and only a subset of properties is usually involved in triggering events.

For example, in the case study (Section 5) – the size of the state space corresponding to the 13 properties and their domains reaches 139968 nevertheless, the actual number of stable states is 24.

Space complexity of PSM is feasible; its upper bound can be estimated as the size of stable states space representation and the PSM transitions representation, and thus it is proportional to $|S| \cdot |P_A| + |S| \cdot |E_A| \cdot \max(|D(p_i)|)$, where $|S|$ stands for a number of possible architecture variants, $|E_A|$ stands for a number of triggering events. In the presented case study the total memory needed by the presented mechanism is less than 2KB.

Since a real-time application generally cannot rely on availability of a dynamic memory allocation, a static assessment of the maximal memory occupied by the event queue (Section 6.2) is an important factor: assuming that (i) each property can be modified by a single primitive component at most once, (ii) each component employs at run-time at most one task, and (iii) RM scheduling algorithm is used [8], the maximal memory size is bounded by

$$\sum_{i \in Co} [2 \cdot T_r / T_i]$$

where $Co$ is the set of indices of all components, $T_i$ is the period of a component with index $i$, and $T_r$ is the period of the task implementing variant switching. Details can be found in [21].

8. RELATED WORK

8.1 Similar Change Propagation Models

A source of inspiration for capturing mode change propagation in property networks was the signal propagation in neural networks [9]. From the others signal propagation models, the most similar one is the Colored Petri Net (CPN) [14]. Although CPN can be used to represent a property network, the design of the net is not as straightforward as in our approach, which is due to different and less specialized semantics of CPN.

Regarding software modes, various formalisms exist for capturing mode in complex software systems (not specifically in software components). In general, all of them consider mode as a first-class entity. These include the State-Charts dialects Mode automata [19], Safe State Machines [2], and the language Esterel [4, 16]. The approaches rely on a data-flow synchronous language featuring discrete time steps in which they re-evaluate a defined data-flow. Contrary to our approach, they require the data-flow to not have cyclic dependencies that would have to be resolved in one time step (i.e. feedback loops have to be realized using “delay” blocks). In our approach, we allow for cyclic dependencies, which are resolved by re-evaluating the property network with the aim of finding a fixed-point represented by a stable state. Another important difference is that these approaches use the data-flow to specify business functionality as well as the mode change, thus effectively blurring the boundary between them. In our approach, we keep them strictly orthogonal. On the other hand, the synchronism in the data-flow language inherently ensures the atomicity (quiescence) required for component switching in a mode change. In our oracle-based solution quiescence is achieved by a mode change protocol cooperating with scheduling. Here, the oracle with its constant time of computation plays a key role.

In general, mode change mechanisms rely on switching predefined variants of software architecture which is very important in time-critical systems. Duration of architecture modification is typically not considered in general models of reconfiguration, adaptation, and self-management [7, 6]. These models include Chemical Abstract Machine (CHAM) [13], graph grammars [17] and typed graph grammars [7] utilizing graph rewriting to specify reconfiguration rules; ADL-based approaches using process algebras [1, 18] or first-order logic.

Finally, in [21] we provide an analysis, why constructing an oracle by encoding properties and functions to a logical formula and using a SAT-solver would be too complex.

8.2 Modes in Other Component Systems

The component systems employing modes include MyCCM-HI [25], BlueArX [15], and Koala [20]. While Koala, not explicitly using “mode”, just allows for switching different types of calls between component variants, BlueArX, having no explicit run-time support for modes, uses handwritten code to call reconfiguration interfaces. MyCCCM-HI [25] introduces mode switching at runtime explicitly [5]. Each component implements a mode automaton
that maintains information about the current mode. The mode change conditions are expressed via boolean expressions determining mode change actions. Technically, each mode automaton is transformed into a control subcomponent that handles the associated component’s reconfiguration (executes the corresponding actions). Based on simulation, the computation related to mode change has to be done at runtime – no pre-computation of variants switching is considered.

9. CONCLUSION
In this paper, we have introduced a theoretical model of event propagation in property networks. Based on the model, the concept of software architectural modes has been formalized in the context of hierarchical component systems. Moreover, a statically precomputed oracle has been introduced, allowing for efficient (in constant time) decision about switching between all potential architecture variants while meeting all the requirements $R_1 - R_4$.

The correctness of oracle reaction to all possible triggering events has been proven. In addition, a proof-of-concept implementation and a non-trivial case-study has been presented and analyzed in detail in terms of the required space and time complexity to show that proposed mechanism is transparent and applicable with a reasonable effort.

As a future work we consider cooperation of multiple oracles in support of mobile and distributed applications.

Acknowledgment
This work was partially supported by the Charles University GAUK project 378111. Grant Agency of the Czech Republic project GACR P202/11/0312 and 201/09/H057 and by the EU project ASCENS 257414. We would like to thank Ondrej Šerý and Honza Kofroň for their valuable comments.

10. REFERENCES