Embedded Systems Resources: Views on Modeling and Analysis

Aneta Vulgarakis Cristina Seceleanu
Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden,
{*aneta.vulgarakis, cristina.seceleanu*}*@mdh.se*

Abstract

The conflicting requirements of real-time embedded systems, e.g. minimizing memory usage while still ensuring that all deadlines are met at run-time, require rigorous analysis of the system's resource consumption, starting at early design stages. In this paper, we glance through several representative frameworks that model and estimate resource usage of embedded systems, pointing out advantages and limitations. In the end, we describe our own view on how to model and carry out formal analysis of embedded resources, along with developing the system.

1. Introduction

Embedded systems (ES) are designed to perform dedicated functions, often under real-time computing constraints. In most cases, they are made of *components* that communicate with each other and the environment via sensors and actuators. The *resources* that such systems use (CPU share, memory, energy, bus bandwidth, ports etc.) are limited in capacity, expensive and (usually) not extensible during the system's lifetime. In contrast to the fixed nature of available resources, software can be subjected to change. To ensure that the combination of components fits on a particular target platform, one needs to be able to estimate the resource consumed by the application software. Further, to facilitate reuse and fast integration of pre-designed components, the ES design techniques should cover systems having minimal resource requirements.

The limited nature of the available resources, especially memory size and computation resources, complicates meeting the real-time constraints. Hence, the ability to quantify and reason about *trade-offs* between various resources, under given technical constraints, is essential. *Prediction* methods for resource usage should be available throughout the whole system's development lifecycle. Access to such information at early stages of design can help the designer to prevent resource conflicts at run-time. This can, in turn, help to decrease the ES' development time and, consequently, reduce development costs. Analysis will then require models of resource usage and theories for composing resource usage models.

Extensive research has been recently devoted to modeling and analyzing ES resource consumption, in component-based design frameworks. Code-level memory estimation for, e.g. Koala- [8, 9] and Robocop- based [12] compositions, as well as higher-level formal approaches [4, 7, 13, 17] aim to establish whether certain resource-related properties hold for a system model.

The main problem of building an ES is correlating its various models of different degrees of detail, which are related via abstraction or refinement. This impacts also on transferring the resource analysis results from one design stage to another. Even so, performing resource consumption analysis at design-time might guide the selection of the appropriate components from existing repositories, when adopting a bottom-up ES design method. Similarly, in a top-down approach, it could help in the correct decomposition of the system's specification into smaller parts (sub-systems) such that the latter could be easier matched by existing sub-systems.

Designing a *predictable* ES amounts, among other things, to establishing that the required resources do not exceed the available resources. For many ES, costs and other constraints make it important to "minimize" resource usage. Here, we advocate a *deployment-oriented* view of ES resource modeling and analysis. For this, we argue that it is important, first, to keep the abstracted hardware model at similar level of detail as the one of the software model, such that the resource-usage analysis can become progressively refined, depending on the design stage; second, we believe that striving for a general formal framework that could be uniformly used throughout the design cycle could be of great help in solving the problem of correlating analysis results.

The variety of approaches existing in the literature indicates the possible difficulty in gathering all resource reasoning in one uniform theory. This calls for a fresh look on resource-aware design methods, based on the lessons drawn from the existing component-based approaches. In the following, we survey some of the current trends in analyzing ES resource accesses, and point out their limitations. We end by describing our view on what is needed to make such methods applicable on a variety of ES, and underline the essential demands of a resource-aware component-based design perspective.

2. Motivating Example

The systematic analysis of the resource consumption of an embedded system must include ways of semantic representation of various types of resources, be they of continuous, monotonic type (like energy), of continuous, non-monotonic type (like memory), or of discrete nature (e.g. I/O ports). A representative analysis goal would be to answer the *feasibility* question: does the composition of the worst-case resource requirements of components stay within the available resources provided by the implementation platform? Checking whether the resource-wise composition is feasible might not be always straightforward.

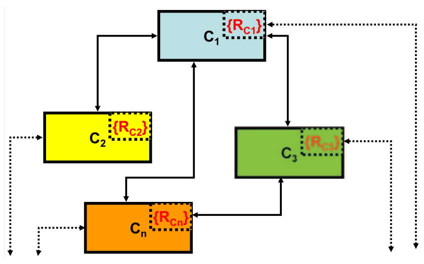

Figure 1. n-component Embedded System with resource annotations.

In practice, it may be often necessary to replace a component with another one having the same functionality, yet using a more sophisticated control algorithm that requires bigger memory resources. Alternatively, if we assume a repository of models, the designer might need, at some point, a refined component model, with modified behavior or more efficiently implementable data structures.

Let us now imagine the following scenario. Suppose that we start building up an embedded system for which we identify the interconnected software components as being $C_1, C_2, \ldots, C_n$ (see Figure 1). The dotted lines represent connections to other possible system components. We also assume that the hardware abstraction provides us with global available resources R. Consider that the computed resource requirement of C_1 is R_{C1}, of C_2, R_{C2} and so on. In Figure 1, the components are annotated with this information.

Suppose now that a different designer wants to use some component B, from the repository, instead of C_1 (for one of the reasons mentioned previously). So, we replace C_1 by B, both functionally and resource-wise. However, it so happens that B needs more resources than C_1 to perform its function: $R_B > R_{C1}$. Intuitively, the resource feasibility test will fail for the new composition, thus preventing us from accommodating B. In order to be able to include B in the system, we need to "fine-tune", in the sense of decreasing enough, the resource requirements of one or more components, for instance, by code-optimization. Then, by rechecking resource feasibility, we should get a positive answer.

A more challenging situation arises if we do not have access to the components implementation. How can we then accommodate B? One could think of trying to change the communication between components, or maybe the allocation of components to hardware units. We would be interested to assess, before deployment, how would any of these design decisions affect the system overall resource consumption. This amounts to finding an appropriate trade-off between different configuration requirements and constraints.

Performing all kinds of analysis of the embedded system's resource usage, starting at an early stage of design, and up to an as close to implementation stage as possible, is extremely desirable. First, it allows for carrying out a potentially large number of design experiments, without increasing cost. Second, it may guide designers in making correct decisions, such as selecting the right components from some repository, choosing among various admissible architectural designs, or transforming a component model into one with less resource requirements.

3. Modeling and Analyzing ES Resources: Representative Current Approaches

3.1 Koala and Robocop: Code-level Analysis

The importance of predicting resource consumption of component assemblies has motivated many researchers to investigate the issue. Compositional ways of estimating the static memory consumption of *Koala*-based embedded system models are already here to help us live up to the resource prediction challenge [8, 9]. Koala [22] is a software component model, introduced by Philips Electronics, designed to build product families of consumer electronics. In the mentioned approaches, resource information is exposed at the component's interface. The *provides* interface defines the operations offered by the component, whereas the *requires* interface defines the operations of other interfaces that the component needs to use. Since in a Koala model all the external functionality that is required by the component needs to go through the "requires" interface, it is somewhat straightforward to estimate the use of the system's resources, such as memory consumption. To estimate a Koala component's static memory consumption, one can assume that a special type of *reflection* provides the interface. For this purpose, Eskenazi et al. introduce the interface *IResource*, which contains the memory consumption demands of each component (see Figure 2). Each of this interface's members corresponds to a particular type of memory. A formula for estimating the memory size of each type of memory is added to the IResource implementation. Since the static properties of a compound component are specified in the reflection interface of its constituents, it follows that it is possible to reuse the respective components in a similar, yet different, composition, without changing its specification.

The above technique supports budgeting, that is, the expected values of the resource consumption of non-implemented components can also be accounted for. An important drawback of the approach is that it can only be used on specific, reduced-size scenarios and concrete component model for which the set of components instantiated in a composition is known before run-time. However, in real-world applications, the situation is much more complicated. If the set of instantiated components changes during run-time, the method will only estimate the memory consumption of the composition for a snapshot of components instantiated at that moment. A lot of experiments, measurements and simulations need to be carried out on the application.

```
<<interface>>
interface IResource
{
  long XROMCODE_size;
  long XROMDATA_size;
  lomg IROMCODE_size;
  long IROMDATA_size;
  long XRAM_size;
  long IDRAM_size;
  long SRAM_size;
  long STACK_size;
  Bool iPresent();
}
```

Figure 2. Example of an "IResource" interface of a Koala component.

Full state-space analysis of system models is most of the time accompanied by combinatorial complexity, as encountered by model checking approaches. In order to avoid such complexity, Jonge et al. [12] introduce a *scenario*-based prediction of *run-time* resource consumption, this time for the *Robocop* component model [18], a variant of the Koala component model. This approach delivers resource estimations for a set of scenarios that represent critical usages/executions of the system. The proposed resource model specifies the predicted resource consumption for all the operations implemented by the services of an executable component. As such, the model contains a number of cost functions that give the operations' costs. There can be multiple cost functions, for each resource. To increase the faithfulness of the prediction, the resources that are claimed and released are specified per operation. Figure 3 shows a revised example of how the service specification is done in [12]. Basically, it specifies service s_1 that requires interfaces I_2 and I_3, and provides interface I_1. Service s_1 implements the operation f that uses the operations g and h from interfaces I_2 and I_3, respectively. In Figure 3, we assume that operation f requires 1200 cpu cycles, without counting the invocations of I_2.g and I_3.h; also, operation f claims 100 bytes of memory before execution, and releases 100 bytes after finishing execution.

```
service s1                 | resource memory
  requires I2              |   claim 100
  requires I3              |   release 100
                           |
provides I1 {              | behavior
  operation f              |   operation f calls:
    uses I2.g              |   I2.g*;
    uses I3.h              |   I3.h}
                           |
  resource cpu             |
    require 1200           |
```

Figure 3. Example of a Robocop component's service specification.

Similar to the reflection interface of the Koala component model, this method is also dealing with static resource consumption, since it is assumed that consumption of resources stays constant per operation. In reality, the former typically depends on parameters passed to operations and previous actions. In addition, the validity of the analysis results still depends on the scenario selection. Moreover, synchronization protocols for analyzing shared resource accesses are not supported. On the other hand, this approach fits very well within current system design practice, such as UML [6], where the dynamics of systems are modeled using scenarios.

The research reviewed so far has been mainly dealing with estimating static memory usage. In many meaningful platform dependent applications, the problem of dynamic resource allocation is acute. Huh et al. [11] address such problem and solve it via *dynamic load balancing* techniques: in case the feasibility test fails, one can either increase the available budget of the current host, or migrate the application to the next best available host. However, this is mostly applicable on distributed, heterogenous systems.

3.2 UML-based Analysis

Low-level, code-driven resource estimates are invaluable when one has access to the implementation of the components, and especially when the components conform to a particular model. Nevertheless, more abstract descriptions of the expected resource usage are also needed in cases of not-yet implemented components or when the designer has to select components from existing repositories, and adapt them to fit the design.

Such abstract descriptions have to not only state what and how many resources are needed, but they should also include information of when and for how long must the resources be available. This extra requirement calls for system *specification* languages. The latter range from fully formal temporal logics [4] and process algebras [19], to the less formal, yet widely used, Unified Modeling Language (UML) [6]. Let us look at some of the UML-based attempts to tackle the analysis of embedded resources.

Baum et al. [2] present a structured approach of describing resource-usage scenarios. For this, they distinguish between two basic classes of resources: *timed-shared* and *space-shared*. Baum argues that any technique for modeling resource-usage scenarios has to consider three description aspects: service requirements, service provision and resource interaction. Service provision captures the characteristics of the services offered by the resource, whereas service requirements describe the resource's demands. Finally, resource interaction links service requirements with provisions. However, such a modeling approach lacks the ability to extend towards a formal description that could provide us with more accurate resource reasoning results.

The UML profile for Schedulability, Performance and Time (UML/SPT) [21] is a framework for modeling concurrency, resources and timing concepts, which eventually produces models for schedulability and performance analysis. From the user's point of view, UML/SPT provides a set of stereotypes and related tag values (i.e., "attributes" in UML 2.0) that can be used by the modeler for the annotation of the model elements and for performing analysis. The core of the profile is the *General Resource Modeling* (GRM) framework. The GRM describes resource types, their static and dynamic interaction with the system, and their management. Each resource offers services for which the effectiveness or quality of service (QoS) is measured. One advantage of the framework is that both static and dynamic resource requirements can be checked against; the disadvantage is the lack of a unique semantical interpretation.

Resources can also be modeled within UML-based simulative environments [1]. For this, Amar et al. extend the UML notation with new stereotypes for performance related items: resource types. The software architecture and the resources that the software components require are both represented in the same capsule diagram, which is split in two parts: the *software* side and the *resource* side.

We exemplify the modeling idea of such an approach through the following example.

Example: A Simple Light Switch System. Consider an ES composed of a display and a fan component, which are turned on/off by the same switch [23]. The software that implements both the light display's and the fan's behaviors utilize memory and processor computational resources.

The software architecture is described by the display and the fan capsules, and the resources that these components require are represented by two more capsules: the memory and the processor. (see Figure 4). The behaviors of the light display, the switch and the fan are depicted in Figure 5. The resource requests issued by the display and the fan application software include the amount of memory/processor needed by each to execute the respective software block.

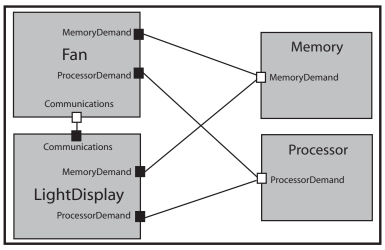

Figure 4. Two-sided capsule diagram.

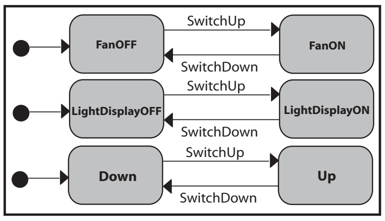

Figure 5. Behavioral diagrams.

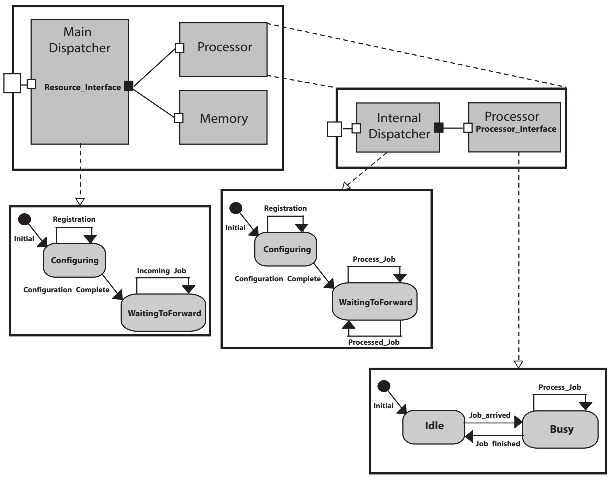

Figure 6. Resource model.

Suppose that the display application needs 300 bytes of memory and a share of 25% processor time to be turned (and while) "on", and just 100 bytes of memory and 5%

processor time to be turned "off" (similar for the software that controls the fan). The resource side is composed by a *Main Dispatcher* and two resource types: memory and processor. In Figure 6 three new stereotypes have been introduced as capsules: a high-level Main Dispatcher, a low level *Internal Dispatcher*, and a *Processor resource*. The state diagrams of these stereotypes can be seen in the lower part of Figure 5. The processor is modeled by a simple state diagram: upon a job's arrival the system is in "busy" state, returning to "idle" when no job is to be served anymore. When the requested memory/processor share, needed for turning on or off the display and/or the fan, has been consumed, a notification is sent to the internal dispatcher and then forwarded to the main dispatcher. The latter checks whether the completed resource request has been satisfied, e.g. whether 300 bytes of memory have been provided to turn on the display, or if the processor utilization has been no more than 5%, for turning the same display off.

Although graphical and intuitive, the above approach is based on a simulative environment, hence one can not entirely guarantee the feasibility of the architecture, but rather provide a partial answer.

3.3 Formal Reasoning on Embedded Resources

Process Algebraic Approaches. In an attempt to unify formal modeling and analysis of ES resources, Lee et al. [15, 16, 17] open a new gate: they propose a family of process-algebraic formalisms that can theoretically account for various resource types. The family relies on algebra of communicating shared resources (ACSR), a discrete-time process algebra that extends classical process algebras with the notion of resource. The starting point of the modeling is the introduction of a resource as a *generic*, *first-class* modeling entity. This comes closer to our wish of being able to correlate various analysis results at different abstraction levels. The authors characterize the resource by a set of attributes, such as timing parameters, probability of failure (π, assumed constant), priority (*pr*, variable), power consumption (*pc*, variable) etc., which capture the resource's behavior. For instance, a class of resources that may experience failures, consume power and whose use can be regulated by properties is captured by the following model:

R: $[\pi :< [0,1] : \mathsf{stat}>, pc :< \mathsf{int} : \mathsf{dyn}>, pr :< \mathsf{int} : \mathsf{dyn}>]$

The authors consider sets of resource classes deemed useful for embedded real-time systems: *serially reusable* shared resources, used to model processor units, *communication resources*, used to model synchronous and asynchronous communication channels, and *multi-capacity* resources that naturally correspond to memory modules. In addition, the general framework is instantiated to several progressively more complex application domains [16].

A first instantiation is Milner's calculus of communicating systems (CCS) [19], in which the resource constraints are equated to just communication constraints between concurrent processes. For example, one can formally enforce model correctness by composing in parallel two processes that send and receive on the same channel. Such an analysis is pretty restricted, as we need to account for other types of resources, as well. Consequently, the authors proceed to extending CCS towards ACSR, which considers time and priorities as resource attributes. An important restriction is the assumption that each action takes exactly one time unit, and that only one process may use a resource during a time step. However, the extension allows for more complex formal analysis, such as correct time reservation of concurrent processes, based on their synchronous execution. The semantic translation of the model gives rise to a transition system that captures the nondeterministic behavior of processes.

Fault-tolerance analysis of embedded real-time systems can be carried out within probabilistic resource failure in real-time process algebra (PACSR), the probabilistic extension of ACSR [16]. Last but not least, memory use is captured as a shared resource among concurrent processes, in the multi-capacity resources algebra (MCSR). Multi-capacity resources are introduced as a new class with two attributes: the capacity of a resource and the memory used by a process during one execution step. Such a rich resource model facilitates reasoning about the effects of reducing the memory use of a process at the expense of its longer execution.

Although the ACSR framework is theoretically rich, the resource analysis is not correlated with the steps towards ES deployment; the verification is independent of the design stage, which makes it difficult to actually use the gained information, when allocating components to the hardware units.

Algorithmic Methods. Quantifying resource usage, such as power consumption, size of message queues, net profit etc., can be done by augmenting *Discrete Time Markov Chains* (DTMCs) with real-valued quantities, called *costs/rewards*, assigned to states and/or transitions [13]. Properties to be verified are expressed in a probabilistic temporal logic (PCTL) extended with reward operators (R). Quantitative verification involves a combination of the traversal of the state-transition graph of the model and numerical computation.

If we consider the light switch example, the properties that could be verified with DTMCs, are:

$$\begin{array}{ll} \mathsf{R}_{\leq 300} & [\mathsf{C}^{\leq 150}] \\ \mathsf{R}_{\leq 5} & [\mathsf{F}\,(\mathsf{light} = \mathsf{off})] \end{array}$$

The first property says that the expected memory consumption within the first 150 time-steps of operation is less than or equal to 300. The second formula states that the expected processor share when the state "light = off" is reached is no more than 5. DTCMs are not really suitable for modeling real-time systems, since there is no notion of real-time, though reasoning about discrete time is possible through state variables "counting" transition steps [13].

A continuous-time approach to analyzing resource consumption is provided by the *Priced Timed Automata* (PTA) [3] framework. PTA are proper extensions of Timed Automata, with cost information on both locations and transitions. Although suited for real-time system modeling, PTA allow mainly continuous, monotonically increasing consumption of resources (e.g. energy) to be modeled and analyzed. How could one then handle non-monotonic resource models (e.g. memory), along with reasoning about, say, energy consumption? The solution might require employing *multi-priced* TA [14], which are PTA with multiple cost variables evolving according to given rates for each location. Even though multi-priced TA are already on the market, a general, unified PTA-based resource model is still missing, as are component-oriented algorithms for verification.

Correct-by-Construction Techniques. The issues of how to deal with reusable resources systematically, and how to convert a program into one requiring less resources may become mind-boggling if systems are complex and heavily resource-constrained. Naiyong and Jifeng [20] address these problems and introduce a resource calculus where comparing two programs that consume/reuse resources is possible. The algebraic laws include program transformation rules that let the designer change the initial program into a less-resource-requiring one. The limitations of the method stays in the fact that the proof-system is not proved complete and program iterations are not considered. Besides, real-time systems can not be covered, since timing information is missing from the models. Even if not component oriented, the approach sheds a light on the meaning of resource-wise program refinements.

A related class of correct-by-construction techniques is focused on the use of *component interfaces* [7]. A well-designed interface exposes exactly the information about a component which is necessary to check for composability with other components. In a sense, an interface formalism is a "type theory" for component composition. As we have seen in the above, recent trends are towards rich interfaces, which expose extra-functional information about a component, like resource consumption levels, besides the functional aspects. Interface theories are especially promising for incremental design under such quantitative constraints, because the composition of two or more interfaces can be defined as to calculate the combined amount of resources that are consumed by putting together the underlying components.

Timed Abstract State Machines (TASM) Approach. *Timed Abstract State Machines* (TASM) is a unified formalism for the specification of functional and non-functional properties of ES [23]. The model is made of two parts, a timed ASM and an environment. Resources are defined at the environment level, such that when a machine executes a step, the updated set of controlled variables contains the step duration and the amounts of resources consumed during the respective step execution. Hierarchical structures and parallel compositions of TASMs are supported, which makes the framework applicable to resource analysis of more complex ES. However, the model seems inadequate for modeling more detailed resource descriptions, since the resource information is a simple annotation, in the form of a real-valued variable assignment. Another limitation is the impossibility to carry out trade-off analysis of conflicting resource requirements.

4. Our Vision of Resource-aware ES Design

In order to be able to synthesize a predictable ES from components and compositions, a *resource-aware* design framework is a must.

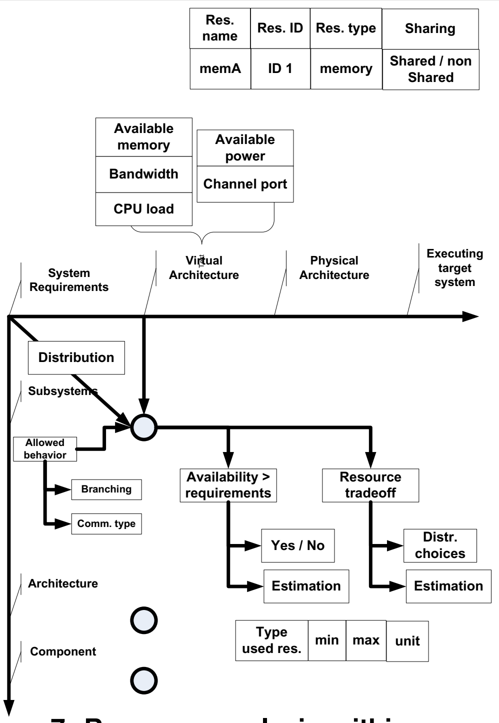

Figure 7. Resource analysis within our proposed ES design flow (courtesy of Severine Sentilles).

Let us assume that the ES under design is real-time, and it is built from components existing at one of the following three levels of granularity (see Figure 7): *subsystem components* (coarse grained with restricted inter-subsystem interaction capabilities), *architectural components* (elements providing an internal decomposition and interconnection structure of subsystems), or *software components* (containers of software with specific interfaces and properties) [10].

Predictability amounts to establishing that the total, worst-case resource usage, in terms of memory, bandwidth, power etc., is within the bounds given by the resources of the selected execution platform. The employed predictability analysis should guide the design and selection of components, as well as the design and selection of the hardware and system software.

Our vision (Figure 7) relies on two pillars: *early stage resource-usage analysis* based on *abstract system models* (at subsystem and architectural levels) and *platform assumptions* (see virtual architecture in the figure), and *later stage* (when a specific system instance to be executed on a specific target is known) *verification* that the assumptions underlying the early analysis are still valid.

The common model-based development approach for ES assumes a "one-way" process, starting from requirements, followed by platform independent system models (PIM), platform specific system models (PSM), and final implementation. As opposed to such a method, we advocate a flexible (iterative) resource-aware analysis framework for real-time ES, which is tightly correlated with the deployment process, from the beginning. The correlation is ensured by the notion of *virtual architecture* on which components can be (virtually) mapped to [10]. The virtual architecture provides an abstraction of the targeted platform, which is gradually added with detail, at the same time with increasing the detail of the system representation. The elements of the virtual architecture (nodes and networks) have resource attributes (memory and power budgets per node, CPU load per node, bandwidth of the communication network, etc.) that allow feasibility analysis, w.r.t resource usage, at different levels of abstraction:

- **Subsystem/Architectural/Software Component-level Resource Analysis (Early stage analysis)**. Suppose that the components $C_1, \ldots, C_n$ of the motivating example (Section2) are either subsystems or software components (SC) mapped onto the virtual nodes. We can assume, at the beginning, a one-to-one mapping. Various design solutions can be investigated for feasibility, by checking whether the resource demands of the subsystems/SCs are smaller than the available ones of the virtual nodes, respectively.

  Early analysis requires a general, unified theoretical framework (like in [3], or [17]) for composing resource-usage models of storage, computation and communication resources. The model should be unified especially for being able to perform *trade-off* analysis between apparently conflicting resource requirements: memory vs. execution time, energy vs. memory etc. If we assume a PTA formal real-time ES model, trade-off analysis would require *multi-objective* model-checking algorithms. However, creating the suitable mathematical framework is work-in-progress, and involves other people than the authors too, hence we just sketch the initial ideas below.

  Within the PTA model, we can encode the notion of a *resource*, by constructing the weighted sum of all the objectives $(c_1, \ldots, c_n)$, as the following cost function: $c = w_1 * c_1 + w_2 * c_2 + \ldots + w_n * c_n$. Here, $c_1, \ldots, c_n$ could describe, for instance, memory-usage cost, energy-consumption cost etc., whereas $w_1, \ldots, w_n$ (weights) could represent the relative importance of $c_1, \ldots, c_n$. The values of the weights are a subjective matter; the way they are chosen depends both on the application and on the analysis goals. For example, if we are designing a heavily resource-constrained soft real-time ES that might tolerate lateness at the expense of quality of service, and are considering trade-offs between memory consumption and (execution) time, we can assign higher weight to memory than to time. To derive the costs, one could apply static analysis techniques on the implementation of a previous version of the software component [5]. Finally, the ES resource-usage would then be described by equation: $\dot{c} = w_1 * \dot{c_1} + \ldots + w_n * \dot{c_n}$.

- **Task-level Resource Analysis (Later stage Verification)**. We now assume some grouping of components $C_1, \ldots, C_n$ into real-time tasks, which are assigned to virtual nodes. Next, virtual nodes are mapped onto physical nodes, according to the resource attributes assumed by the virtual architecture, which are now requirements that the physical architecture must satisfy. The later stage analysis requires a simple yet faithful (abstract) description of the mapping of components, grouped into real-time tasks, onto hardware units.

  If, on top of the hardware abstraction used in the early analysis, we also assume a simple task model (deadline, worst-case execution time, offset, priority), we could roughly estimate resource-usage bounds per tasks, respectively, and their compositions.

The resource analysis process described above is iterative, allowing feedbacks between steps, in case the verification fails. This lets one narrow the space of design solutions, at early stages in the design flow.

We hope that the reader will regard the above arguments as an incentive to looking at more practical ways of incorporating resource information into ES models.

Acknowledgements. The authors thank Séverine Sentilles for lending them Figure 7. This work was performed within the Swedish national strategic research centre PROGRESS, supported by the Swedish Foundation for Strategic Research (SSF) and Mälardalen University.

References

[1] H. Ammar, V. Cortellessa, and A. Ibrahim. "Modeling resources in a UML-based simulative environment". In *ACS/IEEE International Conference on Computer Systems and Applications*, June 2001.

[2] L. Baum and T. Kramp. "Towards a Uniform Modeling Technique for Resource-Usage Scenarios". In *Proc. of Conference on Parallel and Distributed Processing Techniques and Applications*, CSREA Press, July 1999.

[3] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, F. Vaandrager. "Minimum-Cost Reachability for Priced Timed Automata". In *Proc. Hybrid Systems: Computation and Control, 4th International Workshop* (HSCC 2001), LNCS, vol. 2034, pp. 147-161, March 2001.

[4] P. Bellini, R. Mattolini, and P. Nesi. "Temporal Logics for Real-Time System Specification". *ACM Computing Surveys*, 32(1):12-42, March 2000.

[5] A. Bonenfant, Z. Chen, K. Hammond, G. Michaelson, A. Wallace and I. Wallace "Towards resource-certified software: a formal cost model for time and its application to an image-processing example". In *Proceedings of the 2007 ACM symposium on Applied computing*, March 2007.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. "The Unified Modeling Language User Guide". Addison-Wesley, Reading, Massachusetts, USA, 1999.

[7] L. de Alfaro and T.A. Henzinger. "Interface-based design". In *Engineering Theories of Software-intensive Systems*, NATO Science Series: Mathematics, Physics, and Chemistry 195, pp. 83104, Springer, 2005.

[8] E.M. Eskenazi, A.V. Fioukov, D.K. Hammer, M.R.V. Chaudron. "Estimation of Static Memory Consumption for Source Code Components". In *Proc. of Composing Systems from Components Workshop*, IEEE, April 2002.

[9] A.V Fioukov, E.M Eskenazi, D.K Hammer and M.R.V Chaudron. "Evaluation of static properties for component-based architectures". In *Proc. of Euromicro Conference*, pp. 33-39, IEEE, September 2002.

[10] H. Hansson, I. Crnkovic, and T. Nolte. "The World according to PROGRESS". Draft paper, 2008.

[11] E. Huh, L. Welch, B. Shirazi, and C. Cavanaugh. "Heterogeneous Resource Management for Dynamic Real-Time Systems". In *Proc. of the 9th Heterogeneous Computing Workshop*, pp. 287-296, IEEE, May 2000.

[12] M. de Jonge, J. Muskens, and M. Chaudron. "Scenario-Based Prediction of Run-Time Resource Consumption in Component-Based Software Systems". In *Proc. of the 6th ICSE Workshop on Component-based Software Engineering*, pp. 19-24, IEEE, May 2003.

[13] M. Kwiatkovska. "Quantitative Verification: Models, Techniques and Tools". In *Proc. of 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC/FSE), pp. 449-458, ACM Press, September 2007.

[14] K. G. Larsen and J. I. Rasmussen. "Optimal reachability for multi-priced timed automata". In *Theoretical Computer Science*, vol. 390, issues 2-3, pp. 197-213, January 2008.

[15] I. Lee, J-Y. Choi, H-H. Kwak. "A Family of Resource-Bound Real-Time Process Algebras". In *Proc. of the 21st International Conference of Formal Techniques for Networked and Distributed Systems*, Kluwer Academic Publishers, August 2001.

[16] I. Lee, A. Philippou, and O. Sokolsky. "A General Resource Framework for Real-Time Systems". In *Proc. of the 9th Workshop on Radical Innovations of Software and Systems Engineering in the Future*, pp. 234–248, LNCS 2941, October 2002.

[17] I. Lee, A. Philippou, and O. Sokolsky. "Resources in Process Algebra". In *Journal of Logic and Algebraic Programming*, Volume 72, Issue 1, pp. 98-122, May 2007.

[18] H. Maaskant. "A Robust Component Model for Consumer Electronic Products". Philips Research Book Series Volume3, pp. 167-192, May 2005.

[19] R. Milner. "Communication and Concurrency". Prentice-Hall, 1989.

[20] J. Naiyong and H. Jifeng. "Limited Resource Models and Specifications for Programming Languages". *UNU/IIST Report* No. 277, 2004.

[21] Object Management Group. "UML Profile for Schedulability, Perfomance and Time Specification". *Version 1.1, formal/05-01-02*, January 2005.

[22] R. van Ommering, F. van der Linden, and J. Kramer. "The Koala Component Model for Consumer Electronics Software". In *IEEE Computer*,pp. 78-85, IEEE, March, 2000.

[23] M. Ouimet, K. Lundqvist and M. Nolin. "The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems". In *Proc. of the 15th International Conference on Real-Time and Network Systems*, March 2007.