Towards a Time-Triggered Schedule Calculation Tool to Support Model-Based Embedded Software Design

Joseph Porter
Vanderbilt University
Institute for Software
Integrated Systems
Nashville, TN 37203 USA
jporter@isis.vanderbilt.edu

Gabor Karsai
Vanderbilt University
Institute for Software
Integrated Systems
Nashville, TN 37203 USA
gabor@isis.vanderbilt.edu

Janos Sztipanovits
Vanderbilt University
Institute for Software
Integrated Systems
Nashville, TN 37203 USA
sztipaj@isis.vanderbilt.edu

ABSTRACT

Time-triggered architectures (TTA) provide replica determinism in safety-critical distributed embedded software designs. TTA has become a crucial part of many high-confidence embedded paradigms, as it decouples functional concerns from platform timing concerns in system designs. Complex embedded software workflows for safety-critical applications are increasingly managed by model-based design tools, in order to support automated verification and reconcile conflicts between functional and non-functional concerns in designs. We present a prototype scheduling tool (ESched) which calculates cyclic schedules for time-triggered networks. ESched supports the model-based workflow of the ESMoL modeling language and tool suite. Using ESMoL, designers can rapidly iterate through simulating a control design, capturing platform effects in models, generating a schedule (if feasible), and re-simulating the control design subject to the platform model and the computed schedule. ESched specifications include a number of useful platform parameters, and it supports troubleshooting of infeasible schedules by allowing the user to specify partial platform models to solve.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Design

1. INTRODUCTION

The complexity of designing and implementing distributed embedded software for high-confidence (i.e. safety-critical) paradigms has emerged as one of the significant research challenges of our time. The problems involved span numerous domains, which is further complicated by coupling across design concerns in those domains. These challenges are well-documented[11, 6]. Heterogeneous embedded systems design problems fall under the research areas of Cyber-Physical Systems (CPS) and networked control systems (NCS).

In safety-critical systems distribution of functions to multiple processors can introduce nondeterministic behavior, complicating the formal analysis required to demonstrate correct function. Time-triggered architectures[14] (TTA) have emerged as a solution to these kinds of problems. The TTA offers synchronous execution on distributed platforms subject to constraints on tasks (e.g., bounded execution times and logical execution time constraints) and on the platform itself (e.g. time-synchronized processing nodes and support for timed messaging). TTA helps decouple functional design concerns from platform-specific timing concerns.

TTA implementations rely on the computation of a global cyclic schedule for the entire system. Commercial implementations offer tools for this purpose[1] as part of a larger tool suite including cluster design, configuration, and deployment tools. More recently, the research community has spawned a number of projects using model-based development techniques to reconcile functional design techniques with software engineering techniques in embedded software development in high-confidence paradigms. These tools aim to integrate design, verification, and testing under the same umbrella [3, 12, 21], incorporating behavioral models from physical systems, control designs, and implementation platforms [20, 28]. Supporting the workflows of embedded software design tool suites is an important concern for tool developers.

Correct distributed execution in TTA hinges on computation of the schedule. Prior results have shown that constraint logic programming (CLP) techniques can effectively compute schedules for time-triggered systems[22], and scale well to large problems. We have identified two interesting problems when using CLP techniques in model-based workflows:

1. Modeling platform effects – as timing details are known for processing nodes and communication media, they should be incorporated into the scheduling model. How-

ever, overly detailed models can become a burden to create and validate. Models should strike a balance between being overly conservative and being overly detailed.

2. Lack of schedule analysis feedback for designers – CLP solvers either return a valid schedule, or report infeasibility. Designers need useful feedback regarding 'pinch points' in the scheduling constraints in order to adjust timing parameters and determine the effects of timing changes on control designs.

We have developed a prototype scheduling tool based on CLP techniques that includes features to address the problems listed above:

1. The ESched input file allows specification of network topology, functional tasks, and message dependencies. These specifications include computational details relevant to the timing model (e.g. task periods and execution times, and message length) as well as platform timing information (e.g. OS scheduling quanta, bus data rates, and overhead parameters). Specifications also include acceptable maximum latencies between pairs of tasks.

2. Each task or message may be invoked or sent multiple times during a single hyperperiod. We automatically determine instance models and dependencies between them, even for communication between tasks running at different rates.

3. Esched translates instance models to constraint problems for solution, with some modeling improvements over previous techniques.

4. The user can choose to solve the schedule only for partial models (i.e. subsets of the network topology) for troubleshooting infeasible schedules.

5. To improve user feedback for infeasible schedules, we are working on integration of unsatisfiable core extraction techniques to identify minimal subsets of the constraints that are infeasible. This experimental feature is still incomplete.

6. Integration into a model-based tool suite used for designing time-triggered distributed embedded systems[21].

In section 2 we present the problems in greater detail. Section 3 presents the scheduling background as well as the details of our scheduling model. Techniques for troubleshooting infeasible schedules appear in section 4. Section 5 describes a method for testing the scheduler and discusses performance metrics. Integration with the ESMoL modeling language and tools can be found in section 6. Related work and conclusions are in sections 7 and 8, respectively.

2. PROBLEM DETAILS

Model-based design and development have gained great traction in many computational domains. Control designers create models for both physical systems and controllers using

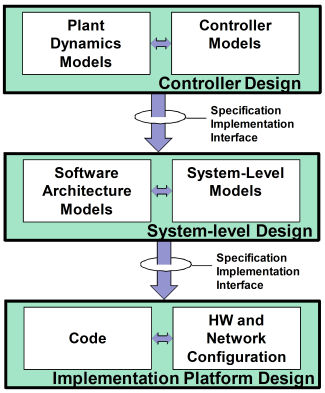

Figure 1: Simplified Model-Based Design Flow

tools like Simulink and Stateflow from the Mathworks[26]. Design verification consists of simulated test cases and use of optimization tools to confirm stability and meet performance goals. Designers adjust parameters or redesign models to meet requirements. Such models (and their supporting tools) correspond to the first stage in Fig. 1 (Controller Design).

In the classical embedded systems design flow, control design models next pass to software developers (Fig. 1, System-level Design). Software designers use models such as UML to specify platform details, code organization, and deployment configurations. Timing is translated from the control specifications, and scheduling analysis is usually done during this stage. Module developers may be given timing budgets to be satisfied.

The final development stage is software integration and testing on the deployed system (Fig. 1, Implementation Platform Design). Engineers or software developers may perform integration and testing. Designed control functions may exhibit new behavior when deployed on distributed architectures, or when code changes are made to address unanticipated problems (e.g., numerical precision, timing issues, etc...). Coupling between the control domain and the software design domain may first appear at this late stage of the process. Designs which failed to account for coupled concerns in earlier stages may face costly redesigns. In any case, late discovery of defects significantly increases cost [17]. In safety-critical NCS designs where defect counts must remain low, early defect detection is crucial.

Model-based design aims to support high-confidence embedded systems design by enabling exchange of information back along the development chain during design iterations. For example, the extraction of useful information regarding problems found during scheduling analysis can enable rapid redesign of of control functions. With respect to Fig. 1, this represents a flow of information against the normal development flow (counter to the arrows) during design iterations. Fig. 2 depicts such a development flow. In order to support rapid model-based redesign, the models at different stages must contain abstractions of the platform execution effects. For example, in scheduling this means that timing uncertainties introduced by the platform should be made available for

use in scheduling models. Even more useful is the ability to isolate scheduling problems and suggest processing rate adjustments to the control designers. This contrasts with the usual yes-no results given by many constraint solvers. In an ideal situation, a scheduling problem could be resolved by a control engineer and an embedded software engineer iterating design parameters and re-running the simulations and scheduling analysis together for an afternoon rather than a costly redesign later on. It should be noted that this is a pattern for a solution – as more kinds of model analysis enter the picture, this process increases in value.

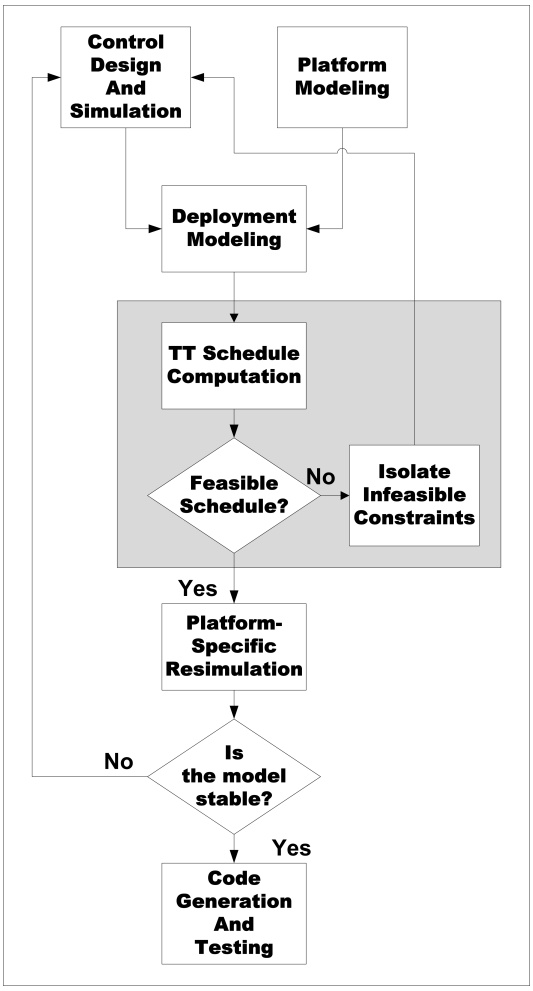

Figure 2: Iterating a control design for platform function. The shaded box in the center highlights the scheduling steps addressed here.

3. SCHEDULING

3.1 CLP Models for Scheduling

In time-triggered communications messages are pre-scheduled on the network, and schedule calculation must ensure that messages and their sending and receiving tasks appear at appropriate times in the schedule, as the protocol does not block for message transfers. For our testing and experimentation we make the following assumptions regarding system models to be scheduled:

- All tasks are strictly periodic, or each task is released exactly on a periodic schedule.
- The worst-case execution time (WCET) is known for each task on a given processor.
- Tasks assigned to the same processor are not preemptive.
- Message buses are shared by multiple processors.
- Messages are not strictly periodic (to ease scheduling), but must be delivered on time.
- Message transfer times are known.
- A feasible bus schedule is not preemptive.
- Messages are broadcast to all nodes on a bus.

All processing nodes execute tasks within the same hyperperiod (repetition window[22]). The hyperperiod is the least common multiple of the periods of tasks in the system. Let H be the hyperperiod length. Each task T_i has maximum duration (WCET) D_i and period P_i. Task T_i runs $\frac{H}{P_i}$ times during a full cycle of the schedule. The individual instances (or jobs) of T_i are denoted $T_{i,j}$ where $j \in [1, ...\frac{H}{P_i}]$. Task instance $T_{i,j}$ has a start time $R_{i,j}$ which is to be determined. The same structure is required for messages. Message M_i has length E_i and instances $M_{i,k}$ having transfer times $S_{i,k}$.

The basic concepts from finite-domain (FD) constraint programming needed for this exposition are covered here in summary only. A constraint problem is modeled as a set of integer variables and a collection of different types of constraints modeling the relationships between those variables. In our case the variables represent the start times of the various task and message instances ($R_{i,j}$ and $S_{i,k}$). The finiteness of FD constraints refers to the fact that all of these variables are positive and have an explicit upper bound. Here all variables initially range over the hyperperiod of the schedule. FD solvers search the solution space by propagating constraints and branching over assignment possibilities. We use the Gecode finite domain constraint solver library[23] within our scheduling tool.

3.1.1 Task Ordering

For each task T_i the instances are ordered by the following set of constraints, representing strict periodicity exactly as in [22]:

$$S_{i,j} = S_{i,j+1} + D_{i,j} \ \forall j \in [1, ...(\frac{H}{P_i} - 1)] \qquad (1)$$

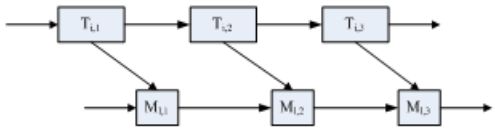

Figure 3: Dependency constraints for instances of a message and its sending task.

For all jobs on the same processor, a global serialization constraint is also imposed to represent non-preemption. The global serialization constraint forces all variable assignments from the given set to be distinct, and to differ by at least a constant value (duration). For efficiency (and model conciseness), global distinctness constraints are used instead of n^2 constraints representing all possibilities[4, 22]. The constant durations are specified for each variable in the constraint, as a pair. For processor p, if I_p is the index set of all task instances on p, then we write

$$\forall i \in I_p, \forall j \in [1, ...\frac{H}{P_i}]\ ser(R_{i,j}, D_i). \tag{2}$$

Here the function *ser* represents the global constraint implementation of the set of distinctness constraints.

3.1.2 Message Ordering

The existing approach to message ordering constraints in [22] explicitly represents the relationships between sender and message instances. To insure that messages were sent between successive sender instances, constraints of the form

$$R_{i,j} + D_i \leqslant S_{l,j}, S_{l,j} + E_l \leqslant R_{i,j+1} \tag{3}$$

are used to capture the requirement that each message instance ($M_{l,j}$) begins after its sending task instance ($T_{i,j}$) and ends prior to the start of the next sending task instance ($T_{i,j+1}$). This captures the logical execution time (LET) constraints as in [10, 16, 9]. For this description we have assumed that messages and their senders occur at the same rate, so they both types of instances are indexed by j. The resulting ordering can be visualized as in Fig. 3.

Message instances are all serialized on a given bus, just as for tasks on a processor. No constraints are imposed on the relationship between message instances and their receivers, as this is handled by latency constraints. We propose the following alternative formulation for message ordering, which is more flexible and hopefully at least as efficient for many useful cases. We will first describe the synchronous case, and then discuss constraints for multi-rate data transfers.

For task T_i sending message M_l (as above), let

$$S_{l,j+1} - S_{l,j} \geqslant P_i - D_i \tag{4}$$

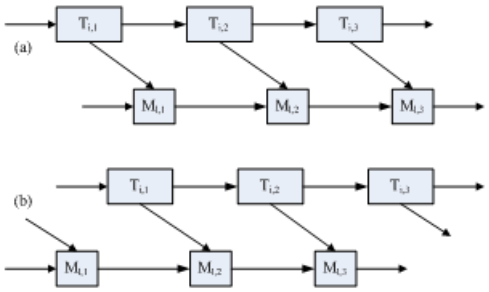

Figure 4: Two possible valid orderings of constraints for instances of a message and its sending task within one hyperperiod.

and

$$(H - S_{l,\frac{H}{P_i}}) + S_{l,1} \geqslant P_i - D_i \tag{5}$$

represent the minimum distance between messages as discussed in [8]. The second expression (5) ensures the minimum distance between the last task in the cycle and the first task of the next cycle. To ensure the ordering as shown in the diagram we serialize all of the instances of both sender and message together using a single global serialization constraint:

$$\forall j \in [1, ...\frac{H}{P_i}],\ ser((R_{i,j}, D_i), (S_{l,j}, E_l)) \tag{6}$$

Notice that this approach represents both valid ways of placing the message instances with respect to the senders (see Fig. 4).

Handling the multi-rate case requires a small modification to the new constraints. We assume that the receiver rate is slower than the sender rate (decimation). This is the only case that makes sense, as interpolation could be more efficiently performed by simply keeping and reusing messages on the receiving end. Suppose $j \in [1, ...J]$, $k \in [1, ...K]$, and that K divides J (written $K|J$). This condition is not strictly necessary, but we leave its relaxation as an exercise for the interested reader. Then we can re-state equations 4 and 5 as follows to include the decimation term:

$$S_{l,j+1} - S_{l,j} \geqslant (P_i \frac{J}{K} - D_i) \tag{7}$$

and

$$(H - S_{l,\frac{H}{P_i}}) + S_{l,1} \geqslant (P_i \frac{J}{K} - D_i) \tag{8}$$

Note that for the simple case $\frac{J}{K} = 2$, this combination of

constraints now represents all four possible proper placements of message instances with respect to the sending tasks, greatly increasing the flexibility of the multi-rate scheduling approach presented in [22].

3.1.3 Latency Constraints

Latency constraints enforce the maximum acceptable response time between the start of a task and the end of a (possibly) different task. Latencies were originally modeled in [22] as a disjunctive constraint for handling the possible wrap-around at the end of the message cycle. The disjunctive constraints apparently created problems for the constraint solver, leading to a two-stage approach to solving the problem [22]. In the two-step approach, the solver is run once without the latency constraints to capture a valid ordering of tasks and messages (to break symmetries). Then the task and message order determined by the solver is used to add the latency constraints.

Latency modeling is greatly complicated by multi-rate tasks and their dependencies. An initial ordering capture may lead to an infeasible schedule, without a clear path to backtrack to a different ordering. Our latency modeling approach does not sacrifice feasibility, but also may not lead to a valid solution. Implicit in selection of a timing bound for a pair of tasks is the set of chains of dependencies between them. These could be captured in the model, but path enumeration can easily lead to a combinatorial explosion of paths. Rather, our approach is to place constraints on the pairs of instances of the starting and ending tasks that try to get them within the specified distance. Then proper ordering must be checked post-facto (i.e. did all of the intervening tasks and messages occur in the proper order). Our argument is that this approach could be verified operationally. For example, re-simulating the control design with the new scheduling order may be a good approach to determine whether time delays introduced by schedule computation have had an effect on the controller requirements. To converge on a working solution, the designers could iteratively generate a schedule for a given design, simulate the control loops running over that schedule, adjust rate parameters (or make other design changes), and repeat (again refer to 2).

The latency model is constructed as follows:

1. Remove redundant constraints – latency specifications that are larger than the period of either the sending or receiving task.

2. For each remaining latency specification, create n^2 reified linear constraints, each requiring that a single sender instance and a single receiver instance fall within the specified time distance. Reified constraints have an explicit boolean control variable.

$$(S_{i,k} - S_{j,l} \leqslant l_m) \Leftrightarrow b_n \tag{9}$$

   In Eq. 9 we see sending task instance $T_{i,k}$, receiving task instance $T_{j,l}$, latency bound l_m, and boolean control variable b_n.

3. Add a constraint on the control variables, so that at least one of the conditions has to be true (Eq. 10).

$$\sum_n b_n > 0 \tag{10}$$

The size of the constraint problem for latencies is not ideal, but the final constraint generally ensures that search will be efficient. Many of the n^2 constraints are infeasible or redundant, so the number could be greatly reduced if necessary. Until this approach introduces significant size or speed issues, we will allow the solver to handle the elimination of redundant and infeasible subsets of those constraints. Tests to date have not shown any problems (on medium-sized models).

3.2 Specification of Scheduling Problems

Listing 1 shows an example of a distributed schedule specification with the following elements:

- **Resolution** (seconds) specifies the size of a single processing tick for the global schedule. This should correspond to the largest measurable time tick (quantum) of the processors in the network. All tasks and messages in the schedule timeline are discretized to this resolution.
- **Proc** specifies a processing node. Parameters are name, processor speed (Hz), and message send/receive overhead times (these default to zero seconds if unspecified). Processor names must be unique.
- **Task** belongs to the most recently specified processor. A task is characterized by its name, period, and worst-case execution time (WCET) (both in seconds). We do not address the manner in which the WCET is to be obtained.
- **Bus** specifies a bus object, characterized by name, transfer speed (bits per second), and transfer overhead (also in seconds).
- **Msg** includes a name, byte length, sending task, and list of receiving tasks.
- **Latency** specifies a timing constraint between two tasks in the system. This could model end-to-end timing constraints between a sensor and an actuator, for example. The maximum latency is given in seconds.

Listing 1: Sample scheduling problem specification.

```
Resolution 2us

Proc P1 100MHz 50us 10us
Task T1 =50Hz 8us
Task T2 =100Hz 10us

Proc P2 100MHz 40us 12us
Task T1 =50Hz 10us
```

```
Task T2 =100Hz 10us

Proc P3 100MHz 50us 12us
Task T1 =25Hz 10us
Task T2 =50Hz 5us

Bus B12 1Mb 0us
Msg M1 16B P1/T1 P2/T1

Bus B23 1Mb 0us
Msg M2 2B  P2/T1 P3/T1
Msg M3 4B  P3/T2 P2/T2

Latency 35us P1/T1 P2/T1
% Latency loop
Latency 100us P2/T1 P2/T2
```

Task and message names are unique only within their scope (processor or bus, respectively). When used globally they are qualified with their scope as shown (e.g. P3/T1). The timing constraints include the various overhead parameters. For example, once the message length is converted from bytes to time on the bus, we add the transfer overhead to represent the setup time for the particular protocol. Engineers must measure or estimate platform behavioral parameters and include them in models for the platform.

3.3 Solution Concerns

The Gecode FD solver provides pre-programmed branching heuristics (as well as capabilities for building custom branchings). We branch first on the start time variables (end times are implied), subdividing intervals at the midpoint. This seems to give a good spacing for tasks and messages in the results. Latency branching comes second, starting with the maximum latency for each specification. No effort is made to optimize the schedule to minimize latencies – the first feasible latency is taken in each case.

3.4 Limitations

ESched has a number of limitations:

- We do not support mode switching. For specification languages that allow mode switching only at the end of the hyperperiod, individual modes can be formulated as separate schedule problems. Finer-grained switching is not supported.
- Time-triggered hardware protocols such as TTP/C or FlexRay subdivide hyperperiods into fixed-length slots and limit message transfers to ease schedule calculation. Compatibility with these protocols would require additional constraints to model the starting and ending times of the slots as well as communication limits (e.g., limiting a task to send a message only once per slot).
- The overhead parameters may be an overly simplistic model for some cases. Each processor and bus pair may have different parameters, depending on the bus type and the protocol used. The scheduling model is discretized to a single resolution for everything in the system. In reality different processors and buses have different timing characteristics. We also do not account for uncertainty due to time synchronization, though it could be included in the overhead parameters.

4. PARTIAL SCHEDULE MODELS

For an infeasible schedule perhaps the most intuitive troubleshooting technique may be the analysis of partial models. If the tasks on a particular processor are not locally schedulable, then adding extra constraints for communication dependencies will not make them more schedulable. The other possibility is to isolate a bus – by selecting all tasks (and only those tasks) which read and write to a given bus, we can make a similar assessment. The results must be interpreted carefully, since local schedulability does not imply global schedulability.

Users of the scheduling tool select submodels by specifying a list of processors and buses (on the command line) to include in the analysis. While building the constraint model, the scheduling tool includes only task and message instances residing on a specified processor, or which communicate via a specified bus. An example will be shown in section 5.

5. SIMULATION RESULTS AND METRICS

Verifying the correct function of a schedule generator can be a daunting task. Below we describe our testing tool, give an example of partial model debugging, and show performance metrics.

5.1 Test Generation

Finding a large variety of realistic scheduling problem instances of significant size presented a testing challenge for our scheduler-building effort. We created a simulator that builds random networks, populates the processing nodes with tasks, and then creates a random network-compatible task dependency graph representing messages. The problem generator parameters are shown below:

- **nodes** – Number of processing nodes in the system.
- **networks** – Number of networks (buses) in the system.
- **nets-per-node** – Max networks connected to a node.
- **max-tasks** – Max tasks running on a node.
- **period** – Fundamental period of tasks(secs). Actual period of a task is either the fundamental period, one half, or one quarter of the fundamental period.
- **msg-prob** – Probability of creating a message between two tasks. This roughly corresponds to the probability parameter for an Erdos-Renyi random graph on each subset of tasks attached to the same network.
- **min-msg, max-msg** – Minimum/maximum message length (bytes). Message lengths are randomly chosen from this interval (uniform distribution).

- **min-util, max-util** – Minimum/maximum task utilization (drawn uniformly from (0.0,1.0)). This parameter controls the chosen task deadline, which is a fraction of the period.
- **sys-res** – System resolution (in seconds).

During generation no effort is made to ensure that the network topology is connected, but randomly adding edges between nodes in different graph components can easily ensure connectivity if desired. The tester uses the Boost graph library[24] for graph structure manipulation.

5.2 Test Cases

For the specification given in listing 1, ESched returned the following schedule:

Listing 2: Sample schedule.

```
Hyperperiod 40 ms

B12:
B12/M1_0 19.932
B12/M1_1 30.092

B23:
B23/M3_0 19.958
B23/M2_0 19.99
B23/M3_1 30.018

P1:
P1/T2_0 4.986
P1/T1_0 9.934
P1/T2_1 14.986
P1/T2_2 24.986
P1/T1_1 29.934
P1/T2_3 34.986

P2:
P2/T2_0 0.044
P2/T1_0 9.994
P2/T2_1 10.044
P2/T2_2 20.044
P2/T1_1 29.994
P2/T2_3 30.044

P3:
P3/T2_0 9.984
P3/T1_0 19.994
P3/T2_1 29.984
```

The schedule lists the release (transfer) times (in milliseconds) for each instance of the tasks and messages given in the spec. For a partial model, we give an option to the scheduler as follows:

Listing 3: Sample schedule.

```
ESched.exe -f test.scs -o test.rslt
  -d P2 -d P1

P1:
P1/T2_0: 4.992
P1/T1_0: 9.994
P1/T2_1: 14.992
P1/T2_2: 24.992
P1/T1_1: 29.994
P1/T2_3: 34.992

P2:
P2/T2_0: 0.044
P2/T1_0: 9.994
P2/T2_1: 10.044
P2/T2_2: 20.044
P2/T1_1: 29.994
P2/T2_3: 30.044
```

Listing 3 shows the schedule only for the tasks on processors P1 and P2, without their message dependencies. This type of partial model is useful for isolating a single processor with an infeasible task set, for example.

5.3 Metrics

Fig. 5 shows execution time for a number of 'fast' test cases. Many feasible scheduling problems resolve very quickly with our current heuristics. A very few infeasible problems resolve quickly as well, while some problems from both classes enter a long search phase. The boundaries of these cases are not well-characterized. ESched needs more rigorous testing and tuning, so these results must be considered preliminary. Maximum memory utilization for long-running problems appears in Fig. 6. The Gecode solver appears to do an excellent job of managing memory consumption, as memory usage levels remain fairly stable during long searches. The results shown here are given for randomly-generated problems of different sizes. Generation parameters were chosen only to keep the problems on the "edge" of solvability, where a fair number of test cases would be quickly solvable, and the others would be more difficult. The resolution parameter can seriously affect the size of the search space, so it must be chosen with care. Ideally it should represent a fundamental scheduling quantum in the physical system, but it may need to be reduced if the searches take too long.

6. SCHEDULING IN MODEL-BASED EMBEDDED DESIGN TOOLS

ESched was created as part of the ESMoL modeling language and embedded software design suite[21]. A modeler imports Simulink/Stateflow models[26] into the ESMoL language, adds architecture, platform design, and deployment concepts, and then synthesizes code for execution on a time-triggered network. The suite includes a portable time-triggered virtual machine[27] which can run on generic Linux or FreeRTOS to implement timed execution of tasks and messages. Execution requires computation of a cyclic schedule for the distributed system.

Fig. 7 illustrates the language integration of the scheduler with the modeling tool using the model-integrated comput-

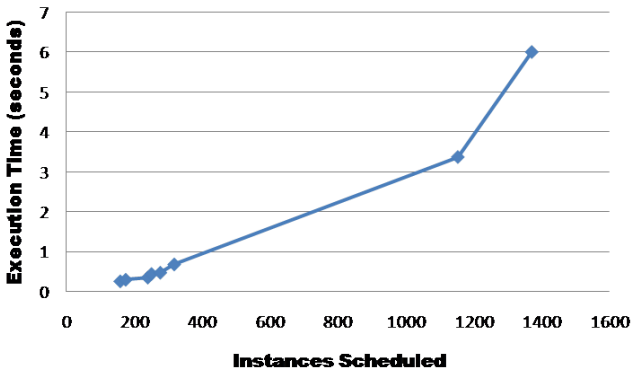

Figure 5: Execution time for feasible instances.

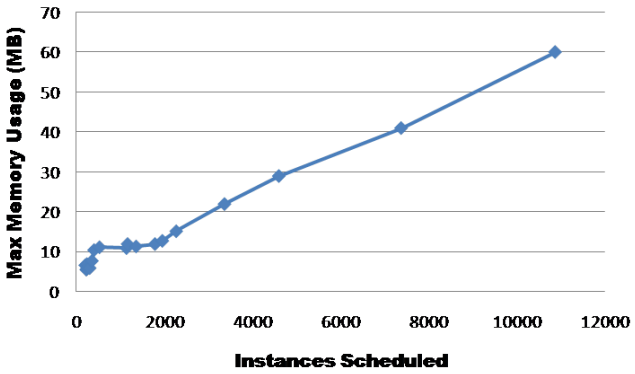

Figure 6: Maximum memory consumption vs. problem instance size.

ing (MIC) tool suite[13]. The MIC tools include a graphical modeling/meta-modeling environment (GME[15]) as well as a model-based graph transformation tool (GReAT[2]). The ESMoL deployment model elements map directly (syntactically and semantically) to the schedule specification language. The tools also parse generated schedules and write the start times back into the model for use in platform-specific code configuration. Extensions currently in development in the tool suite include generation of platform-specific time-triggered simulations in Simulink, using the TrueTime toolkit([20]). The simulation also requires the computation of a schedule.

7. RELATED WORK

A number of modeling frameworks for real-time embedded systems development include some variant of schedule analysis or computation:

- Giotto[10] is a modeling language for time-triggered tasks running on a single processor. Giotto uses a simple greedy algorithm to compute schedules.
- TDL (Timing Definition Language) is the successor to Giotto, and extends the language and tools with the notion of modules (software components)[9]. One

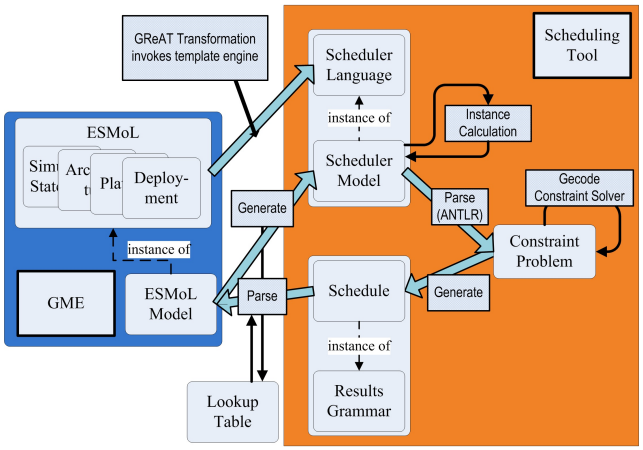

Figure 7: Integration of Scheduling Tool into Model-Based Embedded Software Design Toolchain

version of a TDL scheduler determines acceptable communication windows in the schedule for all modes, and attempts to assign bus messages to those windows[18].

- AADL is a textual language and standard for specifying deployments of control system designs in data networks[12]s. AADL projects also include integration with scheduling tools[25].
- The Metropolis modeling framework[3] aims to give designers tools to create verifiable system models. Metropolis integrates with SystemC, the SPIN model-checking tool, and other tools for schedule and timing analysis.

8. FUTURE WORK

A number of interesting directions lie ahead:

8.1 Expanded Modeling and Semantics

Benveniste et al present an alternative model of computation which relaxes the strict synchrony assumptions of the TTA[5]. The loosely time-triggered architecture (LTTA) guarantees deterministic distributed execution without full clock synchronization between processing nodes. LTTA requires some local timing assumptions for tasks, but further decreases the coupling between timing requirements and functional requirements in the system design. Constraint models for LTTA-compatible nodes and communications media could likely be integrated into the scheduler, for networks that mix these models of computation.

8.2 Solver Efficiency and Correctness

The original order-capturing approach presented in [22] has merit. Reformulating the search as an order search followed by a schedule-time search may lead to more efficient exploration of the search space as well as ensuring that feasible schedules are not missed in the search. This is a form of symmetry breaking, which is commonly used in many scheduling solution techniques.

As mentioned previously, checking the effects of latency constraints may be performed operationally by simulating the effects of the schedule. This is an area of particular interest for the scheduler and the tool suite it supports.

8.3 Unsatisfiable Core Extraction

Unsatisfiable core extraction techniques offer a way to get finer-grained feedback regarding infeasible schedules. For example, if we have analyzed the schedule and identified a particular processor with an infeasible task set, it would be useful to get a minimal set of timing constraints on that processor which could be adjusted to make the set satisfiable (e.g. by increasing the periods or by decreasing the specified WCET).

AMUSE[19] provides an algorithm for extracting minimally unsatisfiable cores, where minimal means a non-unique subset of clauses which could be satisfied by removing any single subclause. Techniques based on SAT modulo theories (SMT) describe techniques for getting minimal unsat cores when including non-boolean formulas in the model[7] (as in our case). Implicit in this sort of extension is a careful consideration of decidability and expressiveness of the scheduling models.

9. ACKNOWLEDGEMENTS

The authors would like to thank Christian Schulte, Guido Tack, Mikael Lagerkvist and others of the Gecode constraint programming community for being willing to answer beginner's questions.

This work is sponsored in part by the Air Force Office of Scientific Research, USAF (grant/contract number FA9550-06-0312). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

10. REFERENCES

[1] Tttech ttplan scheduling tools. http://www.tttech.com.

[2] Aditya Agrawal and Gabor Karsai and Sandeep Neema and Feng Shi and Attila Vizhanyo. The design of a language for model transformations. *Journal on Software and System Modeling*, 5(3):261–288, Sep 2006.

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Paserone, and A. L. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4), April 2003.

[4] N. Beldiceanu and M. Carlsson. A new multi-resource cumulatives constraint with negative heights. In P. van Hentenryck, editor, *CP*, LNCS, pages 63–79. Springer, 2002.

[5] A. Benveniste, P. Caspi, M. di Natale, C. Pinello, A. Sangiovanni-Vincentelli, and S. Tripakis. Loosely time-triggered architectures based on communication-by-sampling. In *EMSOFT '07: Proc. of the 7th ACM & IEEE Intl. Conf. on Embedded Software*, pages 231–239, New York, NY, USA, 2007. ACM.

[6] L. P. Carloni, F. D. Bernardinis, C. Pinello, A. L. Sangiovanni-Vincentelli, and M. Sgroi. Platform-based design for embedded systems. In R. Zurawski, editor, *The Embedded Systems Handbook*. CRC Press, 2005.

[7] A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *Proc. of Tenth Intl. Conf. on Satisfiability Testing (SAT '07)*, volume 4501 of *LNCS*. Springer, 2007.

[8] C. Ekelin and J. Jonsson. Solving embedded systems scheduling problems using constraint programming. Technical Report TR 00-12, Chalmers Univ. of Technology, 2000.

[9] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proc. of the 2005 ACM Conf. on Lang., Compilers, and Tools for Embedded Systems (LCTES '05)*, pages 31–39, New York, NY, June 2005. ACM Press.

[10] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, January 2003.

[11] T. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM: Formal Methods*, volume 4085 of *LNCS*, pages 1–15. Springer, 2006.

[12] John Hudak and Peter Feiler. Developing aadl models for control systems: A practitioner's guide. Technical Report CMU/SEI-2007-TR-014, CMU Software Engineering Institute (SEI), 2007.

[13] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[14] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

[15] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai,

J. Garrett, C. T. IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. *Workshop on Intelligent Signal Processing*, May 2001.

[16] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *Proc. of the 26th Annual Real-Time Systems Symposium (RTSS)*, pages 99–110. IEEE Computer Society Press, 2005.

[17] S. McConnell. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, Redmond, WA, 1996.

[18] A. Naderlinger, J. Pletzer, W. Pree, and J. Templ. Model-driven development of flexray-based systems with the timing definition language (tdl). In *Proc. of the 4th International ICSE workshop on Software Engineering for Automotive Systems*, Minneapolis, May 2007.

[19] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. Markov. Amuse: A minimally-unsatisfiable subformula extractor. In *Proc. of the Design Automation Conference (DAC)*, pages 518–523. ACM/IEEE, June 2004.

[20] M. Ohlin, D. Henriksson, and A. Cervin. *TrueTime 1.5 Reference Manual*. Dept. of Automatic Control, Lund University, Sweden, January 2007. http://www.control.lth.se/truetime/.

[21] J. Porter, G. Karsai, P. Volgyesi, H. Nine, P. Humke, G. Hemingway, R. Thibodeaux, and J. Sztipanovits. Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation. In *Workshops and Symposia at MoDELS 2008, LNCS 5421*, Toulouse, France, To appear 2008. Springer.

[22] K. Schild and J. Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, Oct. 2000.

[23] C. Schulte, M. Lagerkvist, and G. Tack. Gecode: Generic Constraint Development Environment. http://www.gecode.org/.

[24] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*, 2001.

[25] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with aadl. *Ada Lett.*, XXV(4):1–10, 2005.

[26] The MathWorks, Inc. Simulink/Stateflow Tools. http://www.mathworks.com.

[27] R. Thibodeaux. The specification and implementation of a model of computation. Master's thesis, Vanderbilt University, May 2008.

[28] UCB. Ptolemy II. http://ptolemy.berkeley.edu/ptolemyII.