Generating Model Transformations for Mending Dynamic Constraint Violations in Cyber Physical Systems

Sean Whitsitt
University of Arizona
whitsitt@email.arizona.edu

Jonathan Sprinkle
University of Arizona
sprinkle@ece.arizona.edu

Roman Lysecky
University of Arizona
rlysecky@ece.arizona.edu

Abstract

Cyber physical systems (CPSs) by definition involve design constraints addressing the computation and communication necessary to control physical systems. These systems have been modeled using domain specific modeling languages, but some limitations exist in the continued application of such a modeling approach to more complex, or safety-critical, systems. Specifically, it is well known how to formulate constraints in a domain-specific modeling language in order to prevent users from building invalid structures, but existing constraint-based techniques do not take into consideration design requirements that may require analysis in the physical domain (i.e. dynamic constraints). Those analysis results, when interpreted by a domain expert, can inform changes to the model: unfortunately, this "by hand" process does not scale. This paper presents an approach to automate the process of evolving models based on dynamic constraints that are not structurally enforceable into the modeling of CPSs. This new methodology—called dynamic constraint feedback (DCF)—is described herein and demonstrated with specific examples derived from the domain of data adaptable reconfigurable embedded systems (DARES). In DCF, expert blocks are integrated with a modeling language to perform dynamic constraint analysis on system models. The results from these analyses are then used to generate model transformations that can be applied to the source models.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; I.6.5 [*Simulation and Modeling*]: Model Development—Modeling Methodologies

Keywords Model Transformation, Cyber Physical Systems, Dynamic Constraints

1. Introduction

Cyber-Physical Systems (CPSs) are systems where computation, communication, and control interact in a nontrivial way. Overlaps between any two of these three areas have established communities and methods (e.g., real-time distributed systems have computation and communication overlap). However, when all three are considered, exponential increases are often seen in development time, budget, or certification effort [1]. Although cyber-physical systems can be built today (aircraft, home medical devices, automobiles, home automation, etc.), design technologies must improve to enable "scale up" of applications that require validation and verification.

The use of modeling in current CPS paradigms may resist model construction by humans due to the complexity and sheer effort required (e.g., Boeing middleware [2] configurations were frequently 10MB or larger XML files). Abstracting these specifications using "boxes and lines" aids in reducing specification complexity. One limitation, however, is that system behavior and constraints in the continuous domain *are not considered* when evaluating a design's structure. The lack of formality thus requires humans to consider these issues in their head when building models.

The modeling methodology of today lacks the ability to consider constraints *outside the modeling language* as part of the well-formedness of the system. The state of the art utilizes some variant of the object constraint language (OCL) to specify structures that should not be created, or dependencies between model patterns that must be maintained. Although there are individual tools, model-based designers need a reusable technology for CPS constraints that can be more easily evaluated during design time.

This paper refers to constraints that exist outside of the structural context of the modeling language to be *dynamic constraints*. Any constraint that requires analysis to validate is a dynamic constraint. This includes timing, step response, hardware, and other constraints. These constraints can be both explicit and implicit. Explicit dynamic constraints are those that must be specified by the modeler (e.g. that a certain attribute of the system can not exceed some arbitrary bounds). Whereas implicit dynamic constraints are inherent in the domain itself (e.g. it is desirable for a control system to stay stable). Herein, a methodology is described for integrating these constraints into the domain specific modeling languages (DSMLs) of CPSs. Implementing this methodology will enable modelers to automatically validate and maintain designs by keeping them within the bounds of their system's dynamic constraints. Examples are provided from simplified versions of a previously published DSML. For brevity's sake this paper cannot discuss all of the possible example cases or scenarios that are possible. Instead, a few examples have been chosen in order to demonstrate the general conceptual ideas of dynamic constraint feedback (DCF).

At a glance, this DCF methodology involves incorporating analysis blocks into the system to provide feedback on the specified dynamic constraints of the system. These analysis blocks (herein referred to as expert blocks) are "black boxes" that must be developed by an expert in the domain. As such, input and output interfaces have to be developed between the DSML and the expert block. The input interface supplies the expert block with the relevant information from the language and the output interface supplies the DSML with the necessary information to affect a change in the current model. The DSML then generates a model transformation which

can be used to implement that change and the process can be repeated until the expert blocks converge on a final solution for the model. Note that expert blocks are the components of a DSML that suggest changes to the source models. As such, the control logic of any DCF process is in the expert blocks. Note that the DCF process is iterative and models may require several iterations of analysis and transformation in order to be made to meet specified dynamic constraints.

The best analogy to the DCF methodology is Control Theory, where dynamical systems are modeled and their behavior is modified by feedback signals [3]. In this analogy, DCF is the overarching control theory, while the dynamical systems are the models and the feedback signals are the model transformations.

2. Background

The work herein uses the Generic Modeling Language (GME) [4] for the construction of paradigms. The transformations described herein are generated for use with the extension GME, Graph Rewriting and Transformation (GReAT) [5]. The figures shown in this paper will be in the format for these two tools with text added for clarity and brevity.

2.1 DARES

The examples in Section 4 will use the Data Adaptable Reconfigurable Embedded Systems (DARES) domain. A DARES system is one that can be reconfigured in hardware at run time to perform new tasks based on the data that the system needs to process [6]. This allows the system to more efficiently process new data. A model of a DARES system consists of a set of tasks that must be accomplished by the system, the interconnections between them, and a terminal input and terminal output. The connections between the tasks are First In First Out buffers (FIFOs). Each task has several configurations that it can take on depending on the state of the system. Generally there exists a software and a hardware configuration for each state that perform the same function. Ideally, all of the system's tasks would be in hardware since hardware is faster than software in general. However, the restrictions on hardware may prevent all tasks from existing in hardware simultaneously. Therefore, some expert must figure out which tasks should use a software configuration and which should use hardware for any given system state.

2.2 Related Work

Dynamic constraints on CPSs have been described and analyzed before in numerous ways [7, 8]. These papers focus on analyzing and correcting certain dynamic constraint issues within the domain of CPSs. However, none of the current research has focused on integrating these constraints into the context of model driven development. This paper integrates the knowledge from these types of results into the DCF methodology as expert blocks. These blocks are used to perform the analysis of CPSs. The results of that analysis are then fed back into the system and used to generate model transformations that can fix any constraint violations detected by the expert blocks.

2.2.1 Model Transformations

Model transformations have been used to affect models of systems for many years with many available examples [9, 10]. The knowledge that is required for determining what these transformations should be is a part of the expert blocks. The actual implementation and content of expert blocks is independent from the DCF methodology presented in this paper.

Similarly, generating model transformations is not new. There has been much research into the ways in which model transformations can be generated. Notably of these are generating model transformations by example [11, 12] and by demonstration [13]. Both of these approaches are simply ways in which model transformations can be designed and created. These methods can be integrated into the DCF methodology described herein. The important contribution of this paper is not in the actual transformations that are demonstrated, but rather in the application of the methodology. The transformations that are shown in Sections 3 and 4 could be replaced by any other transformation, even ones derived from user examples or demonstrations. The only requirement of the methodology is that an interface exist between the expert block that determines what must be done and the actual portion of the language that generates the transformation.

2.2.2 Models@run.time

DCF should not be confused with models@run.time. Models@run.time extends model based concepts to run-time environments [14], whereas DCF seeks to integrate non-structural constraints into the software development side of model driven development. However, the two are not mutually exclusive. Though the discussion of it is outside the scope of this paper, DCF principles could be applied to models@run.time.

3. Methodology

The methodology will be defined in this section as generally as possible using a DSML called SimpleSim. The reader should keep in mind that specific implementation details (such as the design of the metamodel) are only important to the results of this paper in so far as they provide a generalized implementation example. In Section 4 context is given to the components of SimpleSim to give the reader an idea of how these general ideas may translate to a useful implementation.

The general idea of DCF is that a modeling language integrates with outside expert analysis to modify source models to ensure that those models meet some specified dynamic constraint. The process of analysis and modification to the source models is iterative. It is not expected that the first attempt at solving a dynamic constraint will succeed. Instead, it is expected that the expert blocks will suggest minor, successive alterations that eventually converge the source models toward a solution. This iteration is called the DCF loop.

3.1 A Simple Language

Figure 1 shows the metamodel for the SimpleSim language. At the root level there exists diagrams, constraints, and experts. The experts contain links to the expert blocks that should be run to analyze the models and provide information on how to build model transformations to fix any constraint violations in the models.

The constraints contain values that determine what the dynamic constraints are for the language. The actual meaning and values of the constraints are dependent on the specific domain in question.

Diagrams in the SimpleSim language are where the user draws the actual models. There are three types of element inside of the diagrams: circles, boxes, and triangles. The actual meaning of each of these elements is not important to the generalized understanding of the methodology. In the language an element can be connected to elements of the other two types with no restrictions, but not to an element of its own type. As such, there are six total connection types in the language, two for each unique pair of element types. Each element contains one value, the meaning of which is dependent on the specific domain in question.

3.1.1 Interfaces and Expert Blocks

The modeling language passes the models and constraints to the expert blocks using XML. The expert blocks then read in and translate

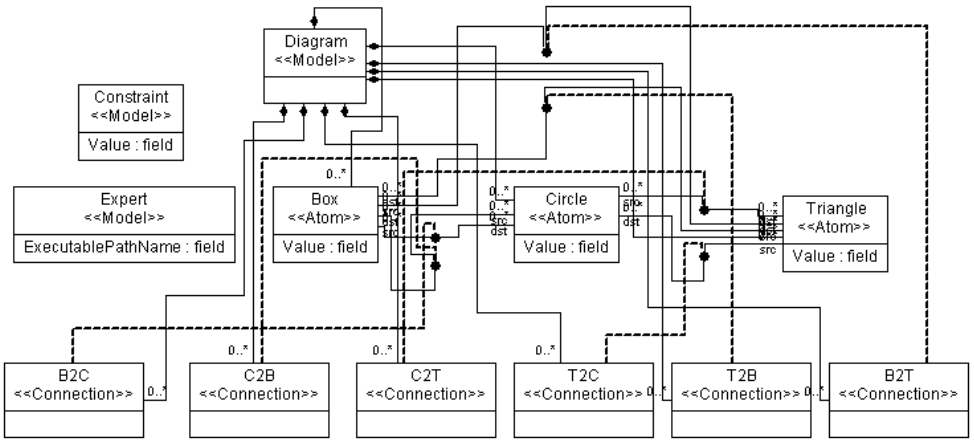

Figure 1: The metamodel for the SimpleSim modeling language.

this information into whatever context they require to perform their analysis. The expert block then performs that analysis and determines what–if any–changes need to be made to the models. At the conclusion of that analysis the expert block must output a transformation or list of transformations encoded in XML format for the SimpleSim language to construct. The SimpleSim language then reads in that XML data and generates the transformations dictated therein.

In general, the functionality of the experts are not considered within the DCF methodology and this paper makes no claims to the efficacy or efficiency of any expert block (even those described herein). Expert examples provided in this paper are mostly conceptual and demonstrative. In general, a modeling language implementing DCF specifies the interface between itself and the expert blocks, but there is no burden on the language designer to implement expert blocks specifically. Instead, experts in the domain must construct the expert blocks that perform analysis. Expert blocks can then be created separately and plugged into the language. Expert blocks must only conform to the interface for input from the language and the interface for the outputs which tell the language what transformations to generate and apply to the source models. It is important to note that the expert blocks only determine which transformations to use, they do not perform the transformations themselves.

3.2 Transformations

The transformations that the language creates based on expert blocks are generated using the skeleton design method [15]. First, a skeleton was made for each component of the transformation. These components include the main GReAT language, the rules of the language, the elements within the rules, the connections between items, the guards that restrict search results within the model, and the attribute mappings that alter the attributes of the elements in the language. Second, these skeletons were generalized in such a way that a simple find and replace string operation can insert required information into the skeletons. These skeletons are then loaded into the modeling language and used to construct and compose the individual parts of a new GReAT transformation. The whole transformation is then constructed as an output artifact of the language. That transformation can then be applied manually to the existing model.

Any type of transformation can be generated, selected, or otherwise used with DCF and any type of method can be used to perform that generation/selection. As such, the details of how these transformations are created is outside of the scope of this paper. For brevity's sake a small set of transformation examples are shown herein. Also, the methodology for creating these transformations can change as well (e.g. the transformations could be generated with model transformation by example methodology instead of using the skeleton design method).

Note that each of these transformation rules can be easily chained together with the skeleton design method to form more complex transformations. For example, the second and third can be chained together to add an element and then connect it to existing items in the model.

3.2.1 Altering Attributes

Figure 2 shows the first generalized example transformation. This transformation simply alters some values within one element, specified by the condition in *Guard_0*, using the value mapping specified in the element *Attribute_Mapping_0*.

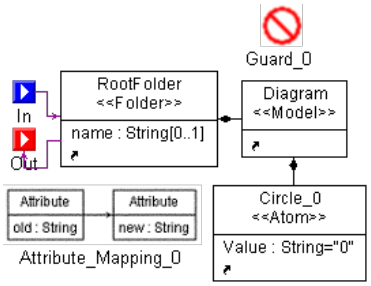

Figure 2: Element attributes can be mapped with a transformation.

3.2.2 Inserting Elements

Figure 3 shows the second generalized example transformation. This transformation inserts a new element into the diagram, specified by *Circle_0*. Note that in the example this item is a circle, but in a general case it could be any type of element that a diagram can contain. The attribute mapping elements specify what the name and value of the new element should be.

3.2.3 Connecting Existing Elements

Figure 4 shows the third generalized example transformation. This transformation connects together three existing elements within the model. The specific elements are determined by the condition in

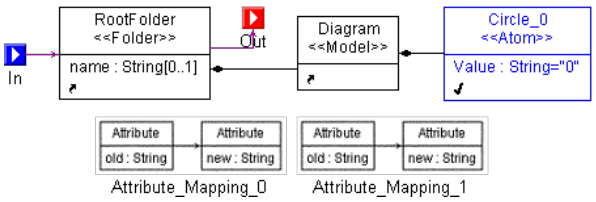

Figure 3: Elements can be inserted into a diagram with a transformation.

Guard_0. The new connections are created with the two elements labeled with the prefix *Connection_Type*. Note that while these specific connections have a type, in general any type of structurally valid connection can be made with this type of transformation.

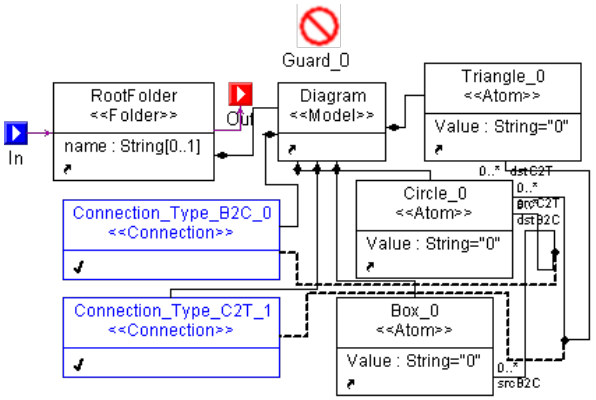

Figure 4: Elements within a diagram can be connected together with a transformation.

3.2.4 Simple Application Example

Figure 5 shows an example in the SimpleSim language where the values are written in the middle of each element. Note that there is no context to this example in this particular case. Figure 6 shows a transformed version of the example where a new element has been inserted and then connected to existing elements based on feedback from an expert block.

To give context to this, imagine that the expert block begins at the leaf nodes (boxes in this example) and traverses the model summing all values that it finds until it reaches a triangle at which point it compares the triangle's value to the summed value. If these values match within some tolerance given by the constraint then it advises no changes. However, note that in Figure 5 that the sum would be 3 and the triangle's value is 5. Since these values are not within 1, the expert must advise a change. In this case it advised inserting a circle between one box and the triangle. This change brings the sum up to exactly 5 because each path to the triangle is summed. All of this context is given based on the expert block used by the system.

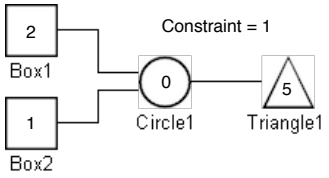

Figure 5: A basic example in the SimpleSim language.

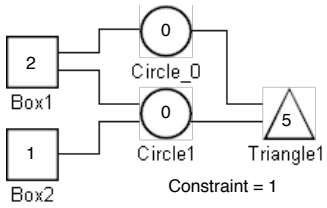

Figure 6: A transformed example in the SimpleSim language.

3.3 Feedback

While this paper does not close the loop on feedback (i.e. automatically generate the new models from the transformations and reanalyze them with the expert blocks), the reader should note that it is only an engineering challenge to do so. Instead, in this implementation users are left to manually apply the generated transformations and call the expert blocks on the transformed models. However, the DCF methodology does allow for expert blocks to alter the models step by step and eventually converge to a solution (assuming a solution exists). It also allows for a model to specify multiple expert blocks to run on a system. However, the danger there is in possible conflicts of interest in the expert blocks.

4. Examples

These examples use the DARES framework discussed in Section 2.1. In these examples, context is now given to the SimpleSim language. Note that nothing has changed in the SimpleSim metamodel except the icons used to display the elements. In these examples circles are FIFOS, boxes are tasks, and triangles are terminal IO.

4.1 Latency/Area Optimizer

As discussed above, one of the goals of DARES is to identify what selection of configurations is optimal for a given data set. As such, in each example the user must specify a latency and an area constraint. In these examples the expert block used attempts to change one selected configuration to bring the system towards a solution that fits with both the area and latency constraints of the system. The user must then implement the generated transformation and run the expert block again to discover whether or not the new system meets their specified constraints.

The latency calculations that the optimizer uses are the same as in the original DARES modeling language. First, a critical path is discovered for the model. This critical path is the longest path from the terminal input to the terminal output that contains no loops. The overall latency of the model is then calculated as the summation of the latencies that lay along this path. The area calculations are simply the summation of all of the area of all of the tasks currently selected to be in hardware.

If the area constraint is not met the expert optimizer attempts to flip a hardware task into software for the best reduction in overall area. After that the optimizer checks the latency constraint. If it is not met, the expert attempts to select the task with the best latency reduction along the critical path first.

4.2 A Simple Dares Example

Figure 7 shows a simple dares example with five tasks. The critical path is marked on the figure with a blue dotted line. To begin, all tasks have software configurations selected, and all tasks have both a software and a hardware configuration available. In this example the area constraint is set to 10000 and the latency constraint is set to 1000. Note that software configurations all have 0 area. Table 1 lists all of the software and hardware configuration information for the example.

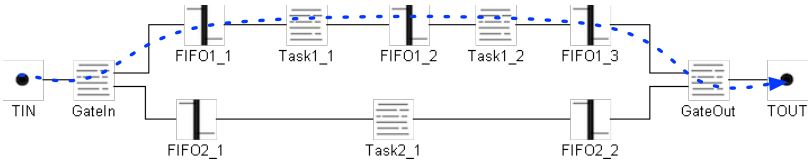

Figure 7: A simple DARES example. The blue, dotted line indicates the critical path for this DARES model.

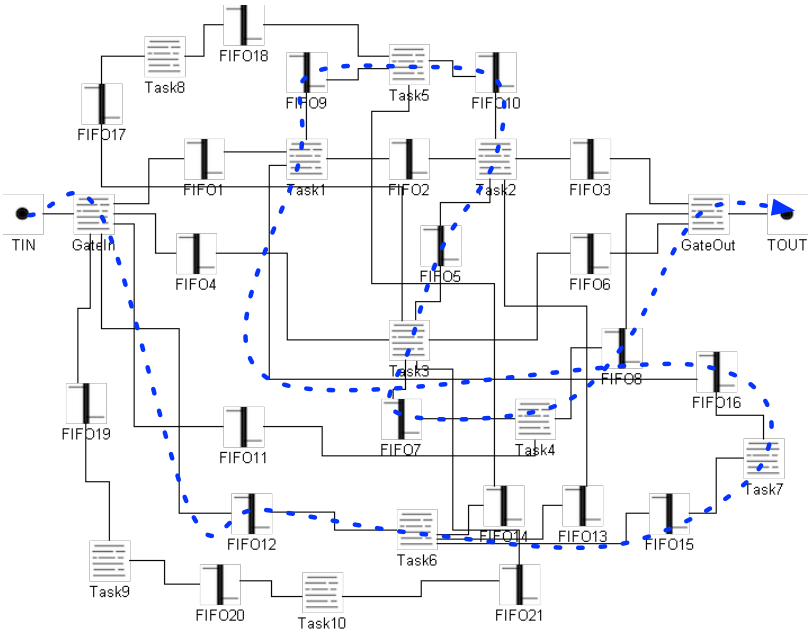

Figure 8: A larger, less human-readable DARES example. The blue, dotted line indicates the critical path for this DARES model.

Task	SW Latency	HW Latency	HW Area
GateIn	1000	80	100
GateOut	1000	80	100
Task1_1	2000	80	100
Task1_2	1000	80	100
Task2_1	500	80	100

Table 1: The software and hardware configuration information for the example in Figure 7.

Repeatedly calling the expert block and running the transformations output by that block eventually converges to a solution after four transformations. Those steps are as follows:

1. Move Task1_1 to HW
2. Move GateOut to HW
3. Move Task1_2 to HW
4. Move GateIn to HW

After these steps are complete the area is calculated as 400 and the latency is calculated as 320, both of which are within the constraints specified. Note that the overall latency of the system may actually be slower since Task2_1 is still in software and has a latency of 500. While this is true, it only indicates an issue with the efficiency and/or correctness of the expert block. If the expert block accounted for this information, it could potentially find more optimal solutions.

4.3 A Larger Example

Figure 8 shows a complex dares example with ten tasks. The critical path is marked on the figure with a blue dotted line. To begin, all tasks have software configurations selected, and all tasks have both a software and a hardware configuration available. In this example the area constraint is set to 400 and the latency constraint is set to 1000. Note that software configurations all have 0 area. Table 2 lists all of the software and hardware configuration information for the example.

Task	SW Latency	HW Latency	HW Area
GateIn	1000	80	100
GateOut	1000	80	100
Task1	2000	100	10
Task2	1000	60	10
Task3	1000	80	100
Task4	2000	80	100
Task5	2000	80	100
Task6	1750	80	100
Task7	2000	80	100
Task8	1000	80	100
Task9	1000	80	100
Task10	500	80	100

Table 2: The software and hardware configuration information for the example in Figure 8.

Repeatedly calling the expert block and running the transformations output by that block eventually results in the following steps:

1. Move Task4 to HW
2. Move Task5 to HW
3. Move Task7 to HW
4. Move Task1 to HW
5. Move Task6 to HW

After these steps are complete a problem is encountered. The area taken up by the hardware tasks is now 410 which exceeds the

constraint value. The expert now starts flipping these tasks back into software and then back into hardware if more iterations of the expert block are run. This indicates to the user that the expert block cannot converge to a solution for the given model with the given constraints. A user in this position has two options available: leave the model as is and deal with the consequences of the unsolvable system, or modify either the system or the constraints. In this particular case, the user knows that latency is a flexible constraint while area is not. If the hardware tasks cannot fit onto the available real estate then the system is not realizable. As such, the user might decide that leaving just Tasks 4, 5, 7, and 1 in hardware is sufficient. In this way it is shown that the methodology of this paper can offer users compromises based on their desired constraints. However, this information could also lead the user to decide that a different hardware chip would be better suited to their system. As such, if the user elects to upgrade their hardware with more area and increase the area constraint to 1000 and continue to allow the expert block to converge to a solution the final system has an area of 720 and a latency of 720 and the following additional steps are generated:

1. Move Task2 to HW
2. Move GateOut to HW
3. Move Task3 to HW
4. Move GateIn to HW

5. Conclusion

This paper describes a methodology for integrating the dynamic constraints of a CPS into the models that represent that CPS. Using expert blocks to analyze those constraints and provide feedback to the system, model transformations can be generated that can help ensure that the system converges toward a valid solution in the context of the user specified constraints on the system. In cases where a solution to the user specified constraints does not exist, the methodology can generate solutions that compromise between conflicting constraints. Using this information users can either redesign their system to better meet their constraints, or relax their constraints such that a solution can be found by the expert blocks.

Additionally, the DCF methodology allows domain experts to disseminate complex CPS analysis tools to non-experts. These tools should enable these non-experts to perform expert level analysis on designs and eliminate many possible dynamic constraint violations in their designs.

5.1 Limitations

Because DCF add additional development steps to the process of creating a modeling language there are additional costs associated with the initial development of a CPS DSML. However, this cost should be made up in the long run by the ability to allow non-experts to create models that are valid to specified dynamic constraints. Also, DCF relies heavily on expert users to define the functionality of the loop that converges the models towards valid solutions. The development of these blocks is not an easy task. However, these difficulties are outside of the scope of the DCF methodology itself.

5.2 Future Work

The next step in this methodology is to actually close the loop on the examples presented herein. This would allow the system to more quickly converge to solutions, and open up the possibility of automatically recognizing conflicts between user specified constraints in the system. Users could then be presented with their options once the system has recognized the conflicts, instead of having to manually discover the conflict locations and manually adapt the system to them. In this case, the methodology could be opened up for more complex analysis through similar avenues as Control Theory has seen. As such, additional future work can also focus on the aspects of expert block development. Additionally, discussion of performance measurements for DCF or the examples herein are outside the scope of this paper for brevity's sake. As such, future work can also expand into discussion of the performance and scalability of DCF.

Acknowledgments

This material is based upon work supported by the National Science Foundation under NSF CAREER CNS-1253334 and NSF CNS-0915010. Additional thanks to Nathan Sandoval.

References

[1] W. Wolf, "Cyber-physical systems," *Computer*, vol. 42, no. 3, pp. 88 – 89, 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.81

[2] D. Sharp, "Real-time distributed object computing: Ready for mission-critical embedded system applications," *International Symposium on Distributed Objects and Applications*, pp. 3–4, 2001.

[3] W. S. Levine, *The control handbook*, ser. The electrical engineering handbook series. Boca Raton (Fl.): CRC Press New York, 1996. [Online]. Available: http://opac.inria.fr/record=b1079196

[4] Vanderbilt University, "Gme 5 user's manual," 2005. [Online]. Available: http://www.isis.vanderbilt.edu/sites/default/files/GMEUMan.pdf

[5] A. Agrawal, "Great: A metamodel based model transformation language," in *18th IEEE International Conference on Automated Software Engineering*, 2003.

[6] A. Milakovich, V. S. Gopinath, R. Lysecky, and J. Sprinkle, "Automated software generation and hardware coprocessor synthesis for data-adaptable reconfigurable systems," in *Proceedings of the 19th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, 2012, pp. 15–23.

[7] B. Hockner, P. Hofstedt, S. Kaltschmidt, P. Sauer, and T. Vörtler, "Design space exploration for cyber physical system design using constraint solving," in *FDL*, 2013, pp. 1–4.

[8] B. Xu and L. Zhang, "Specifying time constraints of cyber physical systems based on clock theory," in *Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering*, ser. CSE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 588–594. [Online]. Available: http://dx.doi.org/10.1109/CSE.2013.93

[9] M. Lawford and W. Wonham, "Equivalence preserving transformations of timed transition models," *IEEE Transactions on Automatic Control*, vol. 40, pp. 1167–1179, July 1995.

[10] J. Dong, S. Yang, and K. Zhang, "A model transformation approach for design pattern evolutions," *IEEE International Conference on the Engineering of Computer-Based Systems*, pp. 80–92, 2006.

[11] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler, "Towards model transformation generation by-example," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, ser. HICSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 285b–. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2007.572

[12] D. Varró, "Model transformation by example," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Springer Berlin Heidelberg, 2006, vol. 4199, pp. 410–424. [Online]. Available: http://dx.doi.org/10.1007/11880240_29

[13] S. Gabmeyer, "Formal verification techniques for model transformations specified by-demonstration," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 390–393. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351756

[14] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[15] S. Whitsitt, J. Sprinkle, and R. Lysecky, "Model based development with the skeleton design method," 2013, pp. 12–19.